UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

20|21

Trabajo fin de grado

Compilador online con Kubernetes

Alejandro Asenjo Gómez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

excelencia Campus Internacional UAM CSIC

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Compilador online con Kubernetes

Autor: Alejandro Asenjo Gómez
Tutor: Alejandro Sierra Urrecho

junio 2021

Alejandro Asenjo Gómez
Compilador online con Kubernetes

Alejandro Asenjo Gómez
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi familia y a mis amigos

“El camino a todo lo grandioso
pasa por guardar silencio”

Friedrich Nietzsche

Agradecimientos

Para empezar, me gustaría agradecer a mi tutor Alejandro Sierra Urrecho su confianza, esfuerzo y atención, que han permitido materializar este trabajo de fin de grado.

Por otra parte, agradecer enormemente a mi familia por su apoyo y sus ánimos, que me han permitido llegar hasta este punto. En especial, quiero agradecer a mi madre por estar siempre junto a mí cuando lo necesito, y ser tan comprensiva.

Por último, agradecer a Pedro Burgos Gonzalo, Iván Bartolomé González y Jose Antonio Cebrían Diz por todos los buenos momentos que hemos pasado, y que han hecho que estos años sean maravillosos.

Resumen

En la actualidad, existen numerosas plataformas de aprendizaje en línea donde aprender a programar. Muchas de ellas disponen de compiladores integrados en sus plataformas, con el objetivo de suministrar a sus usuarios un medio por el que ejecutar código, sin tener que recurrir a configurar sus propios dispositivos. Asimismo, se pueden encontrar multitud de compiladores en línea, utilizables por cualquier usuario con acceso a internet, que permiten a éstos programar desde cualquier dispositivo que tenga instalado un navegador web.

En este trabajo de fin de grado, se pretende desarrollar un servicio de compilador en línea para los usuarios de la plataforma de aprendizaje *ENCODE*. Este servicio estará sustentado a través de una plataforma de contenedores de *Docker*, orquestados a través de *Kubernetes*. Dicha plataforma será instalada en un entorno *On-premise*, de forma que el proceso de instalación sea prácticamente idéntico a las condiciones de desarrollo de la plataforma *ENCODE*, a la cual podría ser integrada en un futuro.

La idea principal es que el compilador permita a sus usuarios ejecutar código desde un navegador web, y aportar una experiencia de uso similar a la que se percibiría al hacerlo en un dispositivo con el *software* de programación necesario para ello. Como primer paso, se creará una versión compatible para programación en C, y progresivamente, en futuros trabajos, se actualizará y mejorará esta versión para añadir lenguajes de programación y otras funcionalidades.

Las tecnologías escogidas para desarrollar la plataforma buscan aportar una alta disponibilidad y una fuerte resiliencia frente a fallos del servicio. Por un lado, la tecnología de contenedores permite reducir el uso de recursos y hacer un uso más eficaz de éstos. Por otro lado, *Kubernetes* automatiza el control y el mantenimiento de todos los contenedores que ejecutarán la aplicación, evitando que un administrador deba hacerlo manualmente, en todo momento.

Palabras clave

Cloud, On-Premise, Compilador en línea, Docker, Contenedor, Clúster, Kubernetes, Kubeadm, Pod

Abstract

Nowadays, there are numerous online learning platforms where you can learn how to program. Many of them have compilers built into their platforms, with the aim of providing their users with a means by which to execute code, without having to resort to configuring their own devices. There are also a multitud of online compilers, usable by any user with internet access, that allow them to program from any device with a web browser installed.

In this Bachelor Thesis, it is intended to develop an online compilator service for the users from the *ENCODE* learning platform, This service will be supported by a *Docker* containers platform, orchestrated via *Kubernetes*. This platform will be installed in an *On-Premise* environment, so that the installation process is practically identical to the development conditions of the *ENCODE* platform, to which it could be integrated in the future.

The main idea is that the compiler allows its users to execute code from a web browser, and provide an use experience similar to what would be perceived on a device with the necessary programming software. As a first step, a compatible version for C programming will be created, and progressively, in future works, this version will be updated and improved to add programming lenguages and other functionalities.

The tecnologies chosen to develop the platform seek to provide the service with high availability and strong resilience to failures. On the one hand, container technology makes it posible to reduce the usage of resources and use them more efficiently. On the other hand, *Kubernetes* automates the control and maintenance of all the containers that will run the application, avoiding that an administatror must do it manually, at all times.

Keywords

Cloud, On-Premise, Online compiler, Docker, Container, Cluster, Kubernetes, Kubeadm, Pod

Índice

Listas

Lista de códigos

Lista de figuras

1

Introducción

Los compiladores en línea son herramientas muy comunes en plataformas de aprendizaje orientadas a la programación. Sus principales ventajas son permitir al usuario ejecutar código sin la necesidad de configurar su dispositivo, y ahorrar un consumo de recursos computacionales, de los que se hará cargo la plataforma.

Aún siendo herramientas con grandes ventajas para los usuarios, presentan grandes dificultades para los desarrolladores. Su principal función, permitir la ejecución de código, supone un riesgo de seguridad que posibilita a un usuario malintencionado tener un canal de actuación directo en la plataforma.

Relacionado con el dato anterior, se presentan diferentes alternativas para desplegar este tipo de servicios como pueden ser contenedores, máquinas virtuales o ambos.

Tradicionalmente, el uso de máquinas virtuales era la solución elegida mayoritariamente, principalmente por simplicidad o porque las aplicaciones suponían un esfuerzo mucho mayor para ejecutarlas en un contenedor. Actualmente, la tecnología asociada a los contenedores ha avanzado enormemente, en especial, el desarrollo de *Kubernetes*, una herramienta de orquestación de contenedores que ha propiciado un cambio de perspectiva. Esto a llevado a que multitud de empresas han tomado en cuenta esta tecnología en su proceso de transformación digital.

En la figura 1.1, podemos ver una muestra tomada a partir de las respuestas de usuarios de la comunidad de *Cloud Native Computing Foundation (CNCF)* en la que se nota un aumento drástico en la proporción de uso de contenedores en entornos de producción empresariales desde 2016. Cabe destacar que en los datos se especifica que el 67 % de los encuestados son trabajadores de empresas con más de 100 empleados, siendo el 30 % de estos, trabajadores de empresas con más de 5000 empleados. La principal conclusión de estos datos, es que los contenedores han logrado obtener la confianza de multitud de empresas y por eso, ha pasado a ser mucho más que una herramienta para facilitar las tareas de desarrollo y pruebas de calidad. [1]

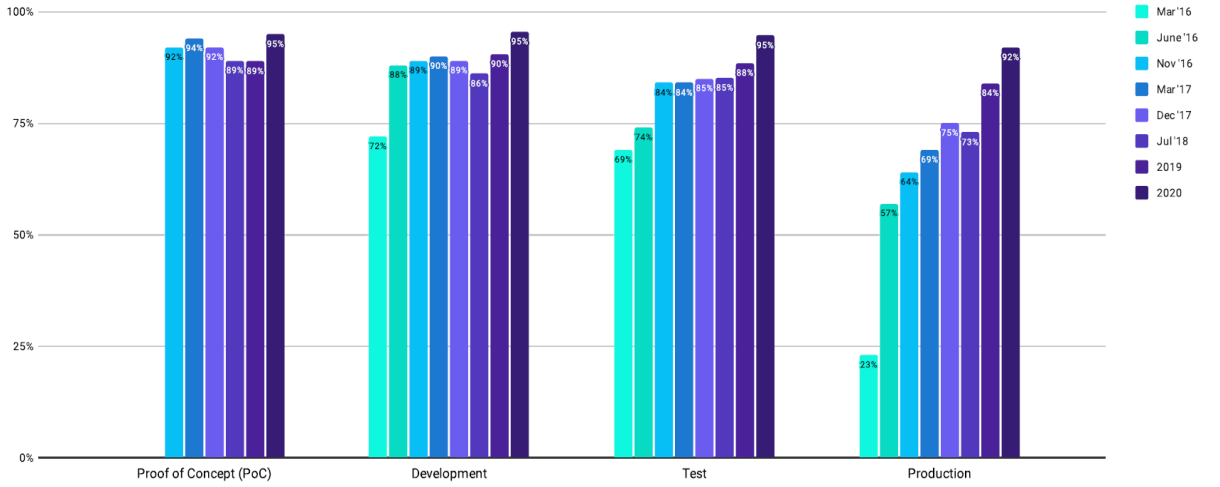

Figura 1.1: Entorno de utilización actual o futuro de contenedores en empresas. *Extraído de* [1]

Por otra parte, Gartner, una empresa de consultoría e investigación, pronostica que *"para el 2022, más del 75 % de empresas a nivel global estarán ejecutando aplicaciones en contenedores en producción, frente a menos del 30 % en 2020"*. [2]

Así, viendo los avances y tendencias tecnológicas actuales, se presenta la posibilidad de implementar esta herramienta alejándose de una arquitectura monolítica y adoptando una estrategia basada en microservicios.

Este trabajo surge con la finalidad de extender los servicios que provee la plataforma *ENCODE*. Esta plataforma fue desarrollada a lo largo del curso académico 2018 - 2019 como objetivo de un TFG. Su objetivo era proveer una plataforma en línea que permitiese el auto-aprendizaje de lenguajes de programación [3]. Posteriormente, se añadieron módulos adicionales que permitían la posibilidad de plantear test de conocimientos a los usuarios y analizar sus resultados [4], o elaborar herramientas de seguimiento del aprendizaje de los estudiantes [5].

En este caso, este trabajo busca dar un primer paso en la creación de un compilador en linea integrado en dicha plataforma. Utilizando, principalmente, tecnologías basadas en contenedores que consigan lograr un buen desempeño y mantengan una gran estabilidad.

1.1. Motivación

Este trabajo surge como una idea para facilitar la labor de aprendizaje que experimentan los usuarios de *ENCODE* durante los cursos de formación.

Hasta el momento, la plataforma consta de un modelo de aprendizaje que enseña al usuario técnicas de programación, pero no le permite experimentar dentro de la propia plataforma, obligándole a

utilizar servicios externos o configurar su propio dispositivo para este fin.

Con la creación de un compilador en línea integrado en la plataforma *ENCODE*, permitiríamos al usuario practicar sus habilidades de programación de la manera más simple posible, y sólo necesitando tener acceso a dicha plataforma.

Por otro lado, añadir este nuevo entorno, abre la puerta a nuevas maneras de monitorizar y orientar el aprendizaje de los usuarios, que se podrían diseñar en un futuro.

1.2. Objetivos

El objetivo principal de este trabajo es proporcionar a los estudiantes de la Escuela Politécnica Superior, que son usuarios de la plataforma de aprendizaje *ENCODE*, un medio por el cual puedan practicar sus habilidades de programación desde cualquier tipo de dispositivo. Para este fin, tenemos que tener en cuenta varios objetivos:

O-1.– Implementación de la aplicación web.

- **O-1.1.–** Desarrollo web *Front-end* que incluirá un editor de código y una pseudo-terminal de Linux interactiva.
- **O-1.2.–** Desarrollo web *Back-end* mediante *Node.js* que gestionará las conexiones de los clientes y sus ejecuciones de código.

O-2.– Creación y configuración del clúster de *Kubernetes*.

- **O-2.1.–** Creación y configuración de los nodos e instalación de software necesario (Docker, Kubectl, Kubeadm...).
- **O-2.2.–** Despliegue de la aplicación.
- **O-2.3.–** Mantenimiento de la aplicación.
- **O-2.4.–** Monitorización del servicio mediante *Prometheus* y *Grafana*.

O-3.– Configuración de acceso al nuevo servicio de la plataforma.

O-4.– Sometimiento de la aplicación a pruebas de funcionalidad y situaciones anómalas.

1.3. Organización de la memoria

La memoria sigue la siguiente estructura por capítulos:

- **Estado del arte.** Se exponen las tecnologías utilizadas y su situación en el panorama tecnológico actual. Por otro lado, se presentan diferentes ejemplos de compiladores en línea, actualmente en producción, que siguen estrategias, en algunos casos similares, y en otros casos, contrarias.
- **Diseño.** Se explican las decisiones de diseño tomadas, incluyendo los requisitos y objetivos del proyecto. Para ello, se desarrolla la arquitectura de la aplicación a constituir, y se analizan las herramientas utilizadas.
- **Implementación.** Se analiza en profundidad el desarrollo llevado a cabo para materializar la solución propuesta

para la aplicación. Desde el desarrollo de la aplicación web para uso del cliente, hasta la creación y configuración del clúster de *Kubernetes*.

- **Pruebas.** Se presentan las pruebas realizadas frente a los casos de uso de la aplicación implementada y se analizan los distintos resultados.
- **Conclusiones y trabajo futuro.** Se ofrece una resolución a partir de los resultados obtenidos sobre la implementación realizada, y se sugieren diversas mejoras a tener en cuenta en un desarrollo futuro de la aplicación.

2

Estado del arte

En este capítulo, se van a presentar varios compiladores en línea, que siguen diseños diferentes, para marcar unas referencias antes de exponer las alternativas tecnológicas a utilizar. Posteriormente, explicaremos en detalle en qué situación se encuentran dichas tecnologías, analizaremos su funcionamiento y confrontaremos distintas opciones que se pueden considerar. En la figura 2.1, se presenta de forma gráfica una aproximación de las tecnologías y alternativas que se expondrán en los siguientes puntos.

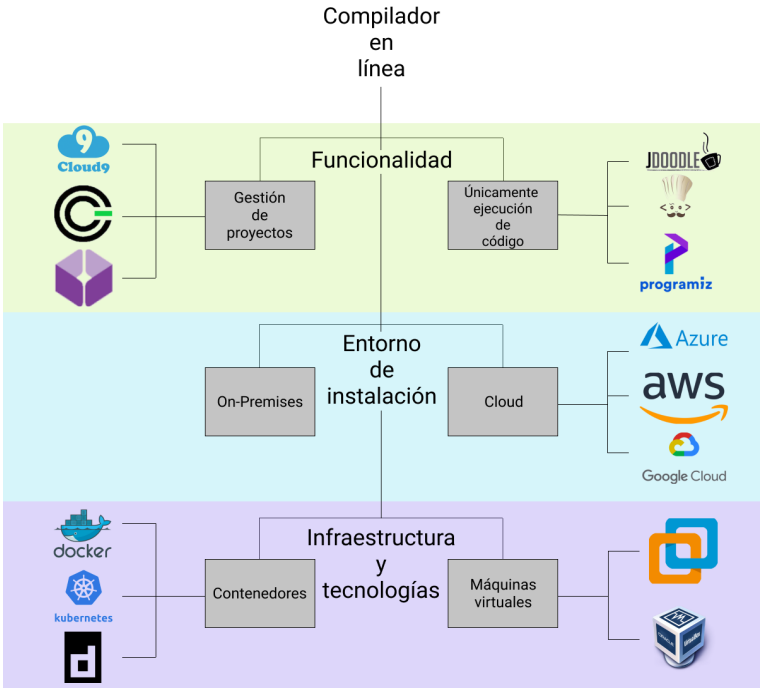

Figura 2.1: Diseño y alternativas tecnológicas a la hora de implementar un compilador en línea.

2.1. Compiladores en línea

Existen multitud de compiladores en línea al alcance de los usuarios. A simple vista, pueden ser divididos en dos grupos, aquellos que permiten a los usuarios almacenar sus proyectos añadiendo esa

funcionalidad de repositorio, y aquellos que permiten ejecutar un solo fichero de código, manteniendo la persistencia de datos durante la sesión de trabajo.

Con esta división, se muestran dos tipos de herramientas que difieren mucho en funcionalidad pero que poseen arquitecturas internas basadas en las mismas tecnologías.

Algunos ejemplos de compiladores que permiten mantener proyectos complejos son: Codeanywhere, AWS Cloud9, Codenvy. Los dos primeros hacen uso de máquinas virtuales para ofrecer servicios a los usuarios. En cambio, Codenvy hace también uso de contenedores Docker para permitir crear espacios de trabajo con distintas configuraciones.

Por otra parte, en el otro grupo señalado, tenemos ejemplos muy dispares en cuanto a funcionalidad, pero la gran mayoría recurren a contenedores para ofrecer sus servicios. Algunos ejemplos son: CodeChef, OnlineGDB, TutorialsPoint, GeeksforGeeks, Programiz, Sphere Engine o Jdoodle, entre otros muchos.

Debido a que el compilador desarrollado en este trabajo no permitirá gestionar proyectos, nos centraremos en las características del segundo grupo. De tal manera, la principal diferencia entre los compiladores es la posibilidad de interactuar en tiempo real con el sistema donde se ejecuta el código y los eventos de la propia ejecución. Para ser detallados, algunos compiladores envían el código a ejecutar al servidor y devuelven como respuesta la salida del programa, pero no permiten ninguna interacción durante la ejecución del programa por parte del usuario.

En contraste, tenemos la alternativa de dar la posibilidad al usuario de interactuar con el sistema que ejecuta el código y así, permitir una experiencia de uso más acorde a ejecutar el código en un dispositivo propio.

Esta característica será la que marque las decisiones de diseño más importantes a la hora de implementar y gestionar los contenedores. Los factores de persistencia de datos y la posibilidad de manipular más de un archivo a la vez serán detalles que marcarán diferencias entre compiladores a nivel de experiencia de usuario, pero a nivel de implementación suponen un impacto mucho menor.

2.2. Cloud vs On-Premise

A la hora de desarrollar una solución tecnológica se suele presentar la disyuntiva de instalar el sistema en un entorno *Cloud*, o en las instalaciones privadas del propietario de la solución, también denominado entorno *On-Premise*. La solución elegida a este respecto, va a marcar las necesidades de recursos económicos, humanos y de infraestructura a lo largo del proyecto. Por tanto, analizaremos en profundidad los distintos beneficios que nos puede aportar cada opción, sin olvidar las consecuencias unidas a éstas.

Para empezar, se presentará el entorno *On-Premise*, que es la solución más tradicional, y se puede

adelantar que será la opción que conlleve un mayor esfuerzo a nivel general. La principal ventaja de este entorno, que es tener completa libertad y control sobre todos los ámbitos asociados al proyecto, supone también la mayor desventaja frente al entorno de *Cloud*. Por una parte, surge una necesidad imperiosa de adquirir recursos de *hardware* físicos adecuados a las necesidades del proyecto, y afrontar su mantenimiento y mejoras. Esto supone que la escalabilidad del servicio a proporcionar supone un problema que resulta complejo y costoso en algunos casos. Por otro lado, se necesita un equipo de profesionales para garantizar el mantenimiento de equipos, referido anteriormente, asegurar la seguridad de los sistemas y datos, y muy probablemente, desarrollar estructuras de datos, infraestructuras de red, etc, que con un entorno *Cloud* se vería muy simplificado.

Si bien es cierto que este entorno supone grandes dificultades para organizaciones que no tienen una infraestructura previa ni recursos humanos cualificados, para organizaciones que sí poseen dichos recursos les puede reportar un ahorro económico, ya que los precios de los distintos proveedores de servicios *Cloud* son elevados. Además, la organización tiene completo control sobre la administración de sus sistemas y aplicaciones, el cual no es comparable con el que permite un proveedor de servicios *Cloud*. Por último, cabe destacar que la conexión a internet es obligatoria para trabajar en un entorno *Cloud*, lo cual puede producir problemas de conexión y latencia que en un entorno de trabajo local no existen.

En el otro lado, tenemos la opción de utilizar un entorno *Cloud* gestionado por un proveedor como pueden ser: Amazon Web Services, Microsoft Azure, GCP o Alibaba Cloud. Antes de empezar a analizar el valor que pueden aportar estos proveedores a una organización, es importante conocer el enorme crecimiento que ha experimentado este mercado en los últimos años.

Según *Bain&Company*, *"El mercado Cloud impulsó aproximadamente el 70 % de todo el crecimiento mundial relacionado con el mercado TI desde 2013 a 2017, y se espera que represente un 60 % del crecimiento del mercado en 2021"* [6] [7]. Esta previsión de crecimiento coincide con los pronósticos de crecimiento que preveía la firma *Gartner*, que manifestaba, *"El gasto mundial en servicios públicos de Cloud se espera que crezca un 18.4 % en 2021 hasta un total de 304.9 billones de dolares, frente a 257.5 billones de dolares en 2020."* [8].

Con estas cifras en mente, se puede comprobar que es la solución preferida para las empresas. Los motivos para que se dé esta situación pasan por permitir a las organizaciones abstraerse de la organización del *software* o el *hardware*, que acaba facilitando mucho el trabajo de ésta. La facilidad para escalar y desplegar un sistema que aporta un proveedor *Cloud* es un gran atractivo, que puede suponer grandes costes si se hace de forma independiente. Por otra parte, los servicios contratados tienen una garantía de disponibilidad total que gestiona y de la que es responsable el proveedor, por lo cual no es necesario tener un equipo de profesionales trabajando para lograr esta situación. Asimismo, para las empresas es un reto abordar la seguridad de sus sistemas y en un entorno de *Cloud* se garantiza una gran seguridad respaldada por grandes equipos de profesionales.

Dicho esto, la parte negativa consiste en justificar los costes de estos servicios frente a los beneficios que aportarán en los distintos ámbitos mencionados. Una mala gestión de la contratación y uso de los servicios puede llevar a costes muy elevados, por lo tanto es importante invertir recursos en gestionar estos servicios de acuerdo con las necesidades de la empresa en ese momento.

Por supuesto, se puede tomar una decisión que implique las dos opciones y organizar los distintos servicios para aprovechar de la mejor manera las ventajas de ambas.

2.3. Contenedores

La tecnología de contenedores se basa en la virtualización a nivel de sistema operativo. Los contenedores comparten el kernel del sistema operativo anfitrión y se encuentran aislados del resto del sistema. Éstos empaquetan una o varias aplicaciones e incluyen todos los archivos necesarios para ser ejecutados. Entre sus ventajas destacan la reducción de los tiempos de implementación y el uso de recursos de manera más eficiente. En la figura 2.2, se pueden ver a nivel superficial las diferencias entre los despliegues en un entorno sin virtualización, uno con virtualización a nivel de hardware, propio de las máquinas virtuales, y el caso de los contenedores.

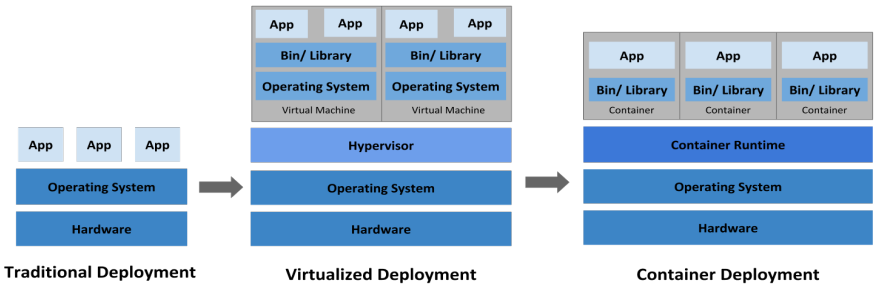

Figura 2.2: Evolución del entorno de despliegue de las aplicaciones. *Extraído de* [9]

Para entender mejor esta tecnología y su evolución debemos retroceder hasta lo que podríamos marcar como su origen, la creación de la llamada al sistema *chroot* que fue introducida en la versión 7 de Unix en 1979. Chroot permitía cambiar el directorio raíz para un proceso y sus procesos hijos con la intención de restringir a usuarios de los que se desconfía el acceso a directorios no deseados. Aún así, no se pretendía que fuera usado para ningún propósito de seguridad [10]. Posteriormente en 1990, se añadió a FreeBSD una utilidad llamada *jail* que reutilizaba la implementación de *chroot*, pero sus funciones no solo se limitaban a restringir el sistema de ficheros, también ofrecía un particionamiento del sistema de archivos, el subsistema de red, los procesos, y eliminaba los privilegios de superusuario, teniendo usuarios independientes del sistema anfitrión [11]. De manera paralela se desarrolló Linux VServer, muy similar a *jail*, que fue añadido al kernel de Linux.

Pero probablemente, los pasos más importantes para la tecnología de contenedores fueron la intro-

ducción de los *namespaces* en 2002, y los *cgroups* en 2007, en el kernel de Linux. Los *namespaces*, permiten dividir los recursos del kernel entre conjuntos de procesos [12], y los *cgroups*, permiten organizar procesos en grupos para limitar y monitorizar sus recursos [13].

Después de la introducción de los *namespaces* y los *cgroups* en el kernel de Linux, en 2008 se conseguiría implementar el sistema *LinuX Containers (LXC)*, el primer gestor de contenedores sobre Linux, que se apoyaría en estas características y proporcionaría al usuario una experiencia de uso parecida a una máquina virtual.

Con la llegada de *Docker* en 2013, anteriormente llamado *dotCloud* [14], se marca cual acabaría siendo la tecnología de preferencia hasta la actualidad. *Docker* es un proyecto de código abierto que permite empaquetar y ejecutar una aplicación de manera aislada en un contenedor. Para este fin, hace uso de las funcionalidades que le proporciona el kernel de Linux, entre otros los *namespaces* y los *cgroups*.

Docker provee un conjunto de herramientas y una API para, entre otras cosas, poder manejar las funcionalidades del kernel y personalizar los entornos de ejecución de cada contenedor. Éste se desarrolla a partir de *LinuX Containers (LXC)*, aunque en la versión de *Docker* 0.9 introducen *libcontainer*, una biblioteca implementada en *Go* para acceder a las APIs del kernel del contenedor, pudiendo prescindir de la dependencia con *LXC* u otros paquetes. [15]

La arquitectura usada por *Docker* consiste en una arquitectura cliente-servidor. El *Docker Engine* es la aplicación principal y se compone de tres partes: [16]

- **Deamon.** Escucha peticiones para la API de *Docker* y administra los objetos de Docker (Imágenes, contenedores, volúmenes y redes).
- **Cliente.** La vía principal por la que el usuario interactúa con *Docker*. Mediante una interfaz de línea de comandos (CLI) y el comando *"docker"* el cliente puede enviar comandos al *deamon* a través de la API.
- **API.** Especifica las interfaces por las que se puede interactuar con el *deamon*.

Por otro lado, tenemos el registro de *Docker* que es donde se almacenan las imágenes. Este registro puede ser configurado como privado, o también se puede utilizar *Docker Hub*, el registro público de imágenes oficial de *Docker* que todos los usuarios pueden usar. Estas imágenes a las que se hace referencia, sirven para crear los contenedores y básicamente, son plantillas personalizables con instrucciones. Estas instrucciones son definidas en un archivo llamado *Dockerfile* que muestra los pasos necesarios para crear una imagen y ejecutarla. La característica principal de las imágenes consiste en que cada instrucción crea una capa en la imagen, de forma que si se realizan cambios en las instrucciones y se reconstruye la imagen, solo las capas modificadas son reconstruidas. Esto hace que sea un proceso ligero y rápido, que le da ventajas frente a otras tecnologías de virtualización.

Para terminar, es importante hacer una pequeña referencia a los datos de crecimiento y uso actuales de *Docker* que han sido posibles mayoritariamente por las contribuciones de empresas como Red Hat, Microsoft o Google, entre otros muchos. En el blog de *Docker* se publicó: *“En julio de 2020,*

Docker Hub ha alcanzado 242 billones de descargas de imágenes. Ese número representa el total de descargas desde que Docker Hub fue creado en junio de 2014" [17]. También, se presenta en el artículo un crecimiento de usuarios y mejoras en las colaboraciones con algunos proveedores de servicios *Cloud*. Estos resultados muestran un futuro muy favorable para *Docker*.

2.4. Kubernetes

Kubernetes fue desarrollado por Google con la intención principal de facilitar a los desarrolladores desplegar y administrar sistemas distribuidos complejos, beneficiándose de la utilización de contenedores. *Kubernetes* se desarrolla a partir de lo aprendido con los sistemas Borg y Omega, de los que Google era propietario. Una de las diferencias principales con los anteriores es que *Kubernetes* fue donado en 2015 a la *Cloud Native Computing Foundation*, lo cual permitió una gran expansión del proyecto. [18]

Muy resumidamente, un despliegue de *Kubernetes* es en esencia un *clúster*. Un *clúster* de *Kubernetes* se compone de un plano de control y un conjunto de nodos que ejecutan aplicaciones en contenedores. Podemos ver los componentes de un *clúster* de *Kubernetes* en la figura 2.3.

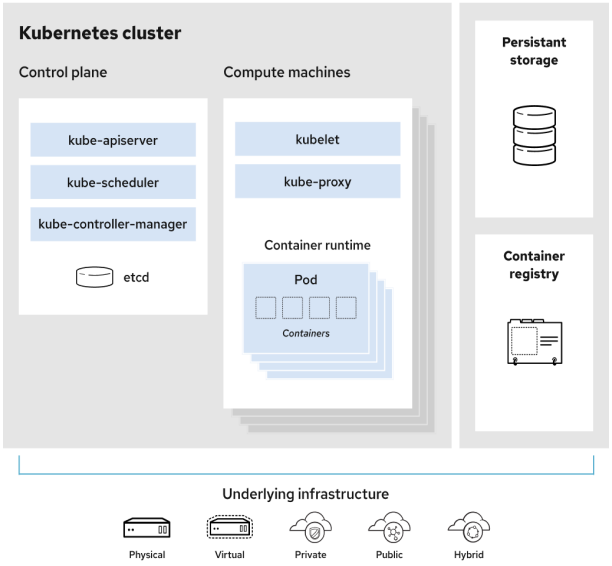

Figura 2.3: Componentes de un *clúster* estándar de *Kubernetes*. *Extraído de* [19]

El plano de control se encarga de controlar el *clúster* y mantiene los datos sobre su estado y configuración. Sus componentes, para simplificar la configuración, se suelen ejecutar en una misma máquina, y en esta no se deben ejecutar contenedores de usuario. Estos componentes son: [20]

- **Kube-apiserver.** Expone una *API HTTP* que permite la comunicación entre los usuarios, las distintas partes del *clúster* y componentes externos.

- **Kube-scheduler.** Planifica en qué nodo deben ejecutarse los *pods* recientemente creados, en base a factores de recursos, especificaciones, etc...
- **Kube-controller-manager.** Ejecuta los procesos de control del *clúster*. Estos controladores son procesos diferentes que se encargan de responder a distintos eventos. Algunos controladores son: controlador de nodos, controlador de replicación, controlador de *endpoints*, controlador de servicios y tokens.
- **Etcd.** Almacena la información del *clúster* de manera persistente y distribuida en forma de clave-valor.

Por otro lado, los componentes que poseen todos los nodos de trabajo son: [20]

- **Kubelet.** Asegura que todos los contenedores se ejecutan en un *pod*.
- **Kube-proxy.** Gestiona las reglas de red en el nodo que permiten la comunicación con los *pods* desde sesiones de red interiores o exteriores al *clúster*.
- **Container runtime.** Es el software responsable de ejecutar los contenedores y administrar sus imágenes. El más popular es *Docker*, aunque también es compatible con containerd, rkt o lxd.

Para finalizar, *Kubernetes* puede instalarse en entornos *On-Premise*, pero también en entornos *Cloud*. El hecho de que los proveedores de *Cloud* más importantes oferten soluciones preconfiguradas para desplegar un *clúster* de *Kubernetes* muestra la gran acogida y popularidad que tiene esta tecnología a nivel empresarial.

3

Diseño

A lo largo de este capítulo, se van a presentar las distintas decisiones de diseño que se han tomado a la hora de constituir el conjunto del sistema. Para ello, se empezará analizando los distintos requisitos que tiene la aplicación y que marcarán un precedente para la toma de decisiones. Posteriormente, se explicará detalladamente la arquitectura del sistema descomponiéndola en dos partes, una primera parte, que es la aplicación web, y una segunda parte, más extensa, que será el *clúster* de *Kubernetes*. La primera parte se expondrá de forma aislada frente a la segunda parte para que en esta última, se encuentren localizadas y justificadas las posibles dependencias entre éstas.

3.1. Requisitos del sistema

En este apartado, se enumerarán los requisitos del sistema que se han considerado necesarios frente a los objetivos presentados en la sección 1.2.

Como aclaración en este apartado, cuando se habla de “aplicación”, se hace referencia a la aplicación ejecutada en cada uno de los contenedores creados en los nodos de trabajo. Por otro lado, cuando se habla de “sistema”, se hace referencia al clúster de *Kubernetes*.

3.1.1. Requisitos funcionales

RF-1.– El usuario puede acceder a una plataforma web con un editor de código y una pseudo-terminal interactivos.

RF-2.– El usuario puede ejecutar código implementado en el lenguaje de programación C.

- **RF-2.1.–** La aplicación muestra al usuario la salida resultante de la compilación y ejecución del código.
- **RF-2.2.–** El usuario puede responder a los eventos I/O de la ejecución de forma interactiva.
- **RF-2.3.–** El usuario puede especificar argumentos de entrada para el programa a ejecutar.
- **RF-2.4.–** El usuario puede detener la ejecución.

RF-3.– El usuario puede manejar un intérprete de comandos (*sh*) que se ejecuta en el contenedor por medio de la aplicación.

RF-4.– El usuario puede administrar ficheros de manera compartida con otros usuarios en las carpetas de su propiedad y el directorio temporal de Linux, *“tmp”*.

3.1.2. Requisitos no funcionales

RNF-1.– La aplicación debe responder a las entradas del usuario en menos de 0.5 segundos.

RNF-2.– Cualquier proceso en ejecución en la aplicación podrá tener una duración máxima de 2 minutos hasta que sea terminado automáticamente.

RNF-3.– Los ficheros creados por el usuario en la aplicación permanecerán en el sistema por un máximo de 2 minutos hasta que sean eliminados automáticamente.

RNF-4.– La aplicación debe permitir la conexión de al menos 10 usuarios de manera concurrente.

RNF-5.– La aplicación es accedida a partir de un dominio personalizado.

RNF-6.– El contenedor de la aplicación no tiene acceso a la red externa.

RNF-7.– El contenedor de la aplicación no puede ejecutarse con permisos de administrador.

RNF-8.– El sistema no permite un consumo de recursos de la aplicación (para cada contenedor) de más de 500 MB memoria, y más de la mitad de la potencia total de un núcleo del procesador.

RNF-9.– El sistema debe mantener la disponibilidad de la aplicación, manteniendo en buen estado todos los contenedores y reiniciando aquellos que hayan sufrido algún error y no sean accesibles.

RNF-10.– El sistema debe gestionar el escalado horizontal de contenedores de forma automática.

RNF-11.– La aplicación es accesible desde navegadores web que tengan habilitada la ejecución de código *JavaScript* y el uso de *cookies*.

RNF-12.– La interfaz de la aplicación está adaptada para ser usada en pantallas con resolución mínima de 1500x720.

3.2. Análisis de la arquitectura

En esta sección, se analizará la arquitectura que constituyen las distintas partes del sistema. Se presentarán dos partes diferenciadas, la aplicación web y el *clúster* de *Kubernetes*. La intención de esto es proporcionar una mayor claridad a la hora de hablar de estas dos partes. A nivel arquitectónico, el *clúster* de *Kubernetes* es el elemento principal que contiene y gestiona todos los elementos existentes y con el que se interactúa desde una red externa. La principal razón para permitir esta división es la posibilidad de desarrollar muchos aspectos de las dos partes de forma independiente, aunque en la segunda parte se harán patentes las configuraciones que están condicionadas y son necesarias para permitir el despliegue de la aplicación web.

3.2.1. Aplicación web

Para comenzar, esta aplicación web será la aplicación que se ejecute en cada uno de los contenedores orquestados por el *clúster* de *Kubernetes*. A grandes rasgos, se trataba de conseguir proporcionar a un número reducido de usuarios concurrentes, la capacidad de ejecutar código en una máquina remota, por medio de una conexión de red, que se realizaría con un servidor web ejecutándose en dicha máquina. Es importante destacar, que esta conexión tendría que ser persistente para proporcionar un entorno interactivo, que respondiese en tiempo real, a la hora de introducir información. Por otro

lado, la intención principal era que fuese lo más cercano posible a ejecutar código en tu propio dispositivo, por lo cual, la integración de un pseudo-intérprete de comandos en la plataforma para mostrar y enviar información era una decisión de diseño imprescindible. A continuación, se explicará en detalle que decisiones de diseño se toman a partir de esta idea básica.

Como se puede ver en la figura 3.1, en el primer momento que el usuario accede a la aplicación, se produce una conexión persistente entre el usuario y el servidor. Además, en el lado del servidor se crea un interprete de comandos *sh*, con el que podrá interactuar el usuario desde su navegador.

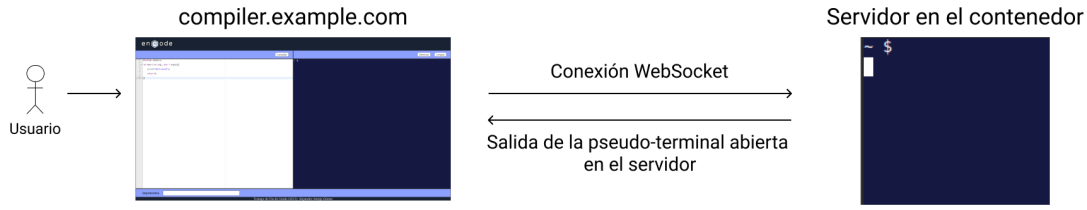

Figura 3.1: Respuesta de la aplicación frente al acceso de un usuario.

Estas dos características, suponen el núcleo principal en el que se sustentan las funciones de la aplicación. Al tener tanta importancia, condicionan profundamente la elección de la tecnología para la implementación del servidor. En este caso, basando la decisión en torno a las características y las herramientas disponibles de la tecnología, la elección tomada es utilizar el entorno de ejecución de *JavaScript*, *Node.js*.

Node.js es una tecnología que permite crear aplicaciones de red escalables. De esta forma, las conexiones simultáneas de los distintos usuarios que accedan a la aplicación serán gestionadas de forma eficiente. Como ventaja frente a otras tecnologías que siguen modelos de concurrencia, *Node.js* utiliza un solo hilo de ejecución para gestionar las conexiones de todos los usuarios, lo cual repercute positivamente en el consumo de recursos de la aplicación [21]. Por otro lado, *Node.js* posee *Node Package Manager (npm)* [22], un excelente gestor de paquetes que nos facilita la instalación de módulos y es ideal para gestionar la instalación de éstos al crear la imagen del contenedor para esta aplicación.

Los módulos utilizados para lograr la funcionalidad anterior serán:

- **Express.** Framework para aplicaciones web que proporciona un conjunto de herramientas para servidores *HTTP* [23].
- **Socket.IO.** Biblioteca que permite la comunicación bidireccional, en tiempo real y basada en eventos entre el navegador y el servidor. [24].
- **Xterm.** Componente *Front-end* que permite integrar terminales en los navegadores de los usuarios [25].
- **Node-pty.** API que permite generar pseudoterminales en procesos con descriptores de fichero que permiten lecturas y escrituras [26].

Por otra parte, a la hora de configurar el sistema donde se ejecutará el código por medio de la pseudo-terminal, es importante crear un usuario sin privilegios de administrador para evitar acciones

indebidas por parte del usuario, y de esta manera, también limitamos la gestión de archivos a la que el usuario está autorizado. Además, hay que reducir el área de maniobra a la que un posible atacante puede recurrir, lo cual, lleva a utilizar una distribución de Linux simple y segura, como es *Linux Alpine*. Esta distribución es muy ligera, por tanto, este ahorro de recursos es una ventaja a la hora de escalar horizontalmente el número de contenedores disponibles.

Por último, se van a presentar las tres principales acciones que podría llevar a cabo un usuario en la aplicación. En primer lugar, como se puede ver en la figura 3.2, el usuario puede enviar al servidor el código que haya escrito en el editor de código y los argumentos de ejecución que haya especificado en el apartado disponible para ello, y éste será compilado y ejecutado, para posteriormente mostrar las salidas que se hayan producido en la terminal.

Figura 3.2: Acción de compilar código por el usuario en la aplicación web.

En segundo lugar, como se muestra en la figura 3.3, el usuario puede parar la ejecución de su programa y obtener una nueva terminal.

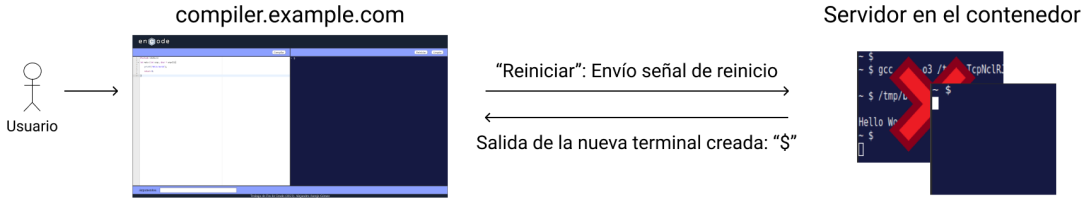

Figura 3.3: Acción de reiniciar la terminal por el usuario en la aplicación web.

En tercer lugar, como se aprecia en la figura 3.4, el usuario puede introducir texto en la interfaz correspondiente a la terminal, y éste es enviado a la terminal del servidor devolviendo el resultado.

Figura 3.4: Acción de introducir texto en la terminal por el usuario en la aplicación web.

3.2.2. Clúster de Kubernetes

El *clúster* de *Kubernetes* engloba todo el sistema a construir y se encarga de gestionarlo. Esta gestión consiste en numerosos factores que harán que el sistema tenga una alta disponibilidad, sea resiliente frente a fallos y pueda gestionar las cargas de trabajo. Toda esta gestión se llevará a cabo de manera automática y se sustentará en la configuración que se establezca por medio de las herramientas de *Kubernetes*. Por supuesto, *Kubernetes* es la plataforma ideal para este fin, ya que entre sus funciones principales están el descubrimiento de servicios, el balanceo de cargas de trabajo y la auto-comprobación del estado de los contenedores, funciones imprescindibles para crear un sistema altamente disponible.

El estado de un *clúster* de *Kubernetes* es representado a partir de los llamados objetos de *Kubernetes*. Estos son entidades persistentes dentro del sistema de *Kubernetes* que el usuario puede gestionar a partir de la *Kube-apiserver*. Los objetos pueden describir las aplicaciones que se están ejecutando, y con qué condiciones de recursos o políticas de comportamiento. Éstas dos últimas, condicionan el funcionamiento de las aplicaciones, como pueden ser estrategias de reinicio, políticas de red, limitación de uso de memoria y procesador, etc, entre otros [27].

Kubernetes facilita la automatización de la organización de todos los objetos de *Kubernetes* por medio de una configuración de tipo declarativo, proporcionada por el usuario. Mediante esta configuración, *Kubernetes* compara el estado actual, con el estado deseado propuesto por el usuario para modificar los objetos de tal forma. Así, casi todos los objetos tienen dos apartados de configuración, el apartado *spec*, que muestra el estado deseado, y es configurado por el usuario para proporcionar un conjunto de características al recurso; y el apartado *status*, que es actualizado por el sistema de *Kubernetes* para mostrar el estado actual del objeto [27].

Una vez se conoce cómo funciona la configuración de *Kubernetes*, el primer paso es crear la infraestructura que hospedará las distintas partes del *clúster*. Como la plataforma *ENCODE* dispone de un servidor dedicado y el coste de utilizar un proveedor *Cloud* sería elevado frente a los beneficios que pudiera reportar, el entorno elegido para desplegar la infraestructura será un entorno *On-premises* que sería gestionado por un administrador. En la figura 3.5, se pueden ver los diferentes elementos que sería necesario establecer.

Como se puede comprobar, dentro del servidor se crearán varias máquinas virtuales. Una de ellas, actuará como el plano de control del *clúster* de *Kubernetes* y las demás, actuarán como nodos de trabajo vinculados al plano de control. Para lograr esta interconexión entre máquinas virtuales se configurará la red interna donde se encuentra el servidor, de tal forma que las máquinas virtuales tengan direcciones ip únicas y sean accesibles entre ellas. Por otro lado, el elemento externo al *clúster* de *Kubernetes* será la única forma de acceder a éste. Este elemento es un *proxy* inverso que proporcionará acceso al *clúster*, pero éste último, no podrá ser accedido directamente por un usuario desde una red externa a la que se encuentra. Este diseño de red responde a segurizar el *clúster* a la hora de posibles

ataques de denegación de servicio o intrusión.

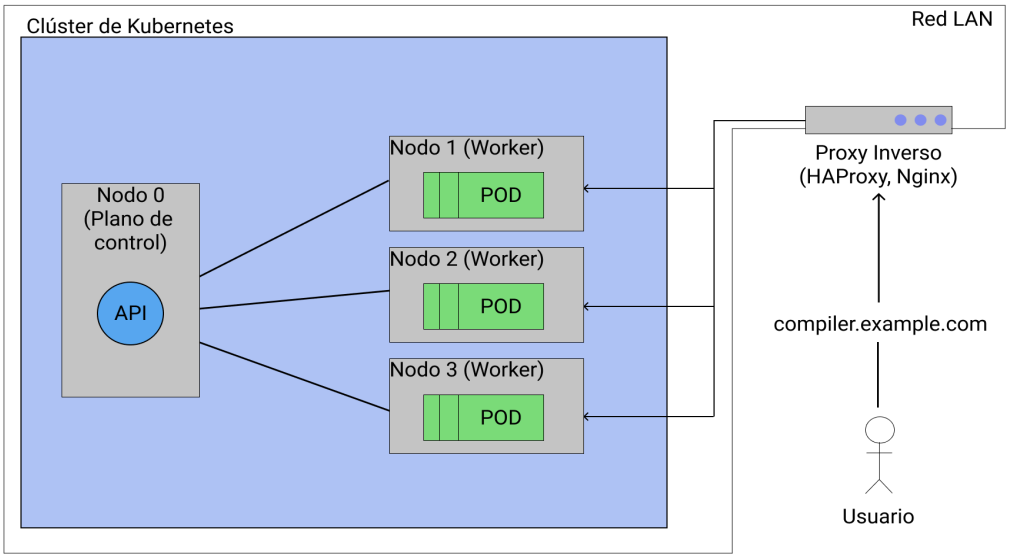

Figura 3.5: Organización básica del *clúster* de *Kubernetes* en el servidor local.

El siguiente paso, es elaborar la configuración de los distintos objetos de *Kubernetes* y así, establecer el estado en el que se desea que el *clúster* de *Kubernetes* opere. La configuración a realizar tendrá en mente los siguientes objetivos:

O-1.– Despliegue de la aplicación de forma que sea altamente disponible y resiliente a fallos.

- **O-1.1.–** Configuración de replicación de la aplicación y su escalado horizontal.
- **O-1.2.–** Configuración de disponibilidad y limitación de uso de recursos.
- **O-1.3.–** Configuración de pruebas de estado de la aplicación.

O-2.– Exposición de la aplicación a partir de un dominio único.

O-3.– Mantenimiento de la aplicación mediante acciones programadas.

- **O-3.1.–** Liberación de recursos dentro de la aplicación.
- **O-3.2.–** Reinicio de contenedores de manera programada.

O-4.– Posibilidad de monitorizar los recursos utilizados.

Por otro lado, para que la aplicación pueda ser desplegada en los contenedores, se necesitará instalar *Docker* en los nodos de trabajo y construir la imagen de la aplicación de forma local en cada nodo. De esta forma, *Kubernetes* tendrá acceso a las imágenes locales precargadas y se producirá un ahorro de tiempo al prescindir del proceso de descarga de la imagen de un posible repositorio, público o privado.

En la figura 3.6, se puede ver de forma simplificada la relación entre *Kubernetes* y *Docker* en los nodos de trabajo.

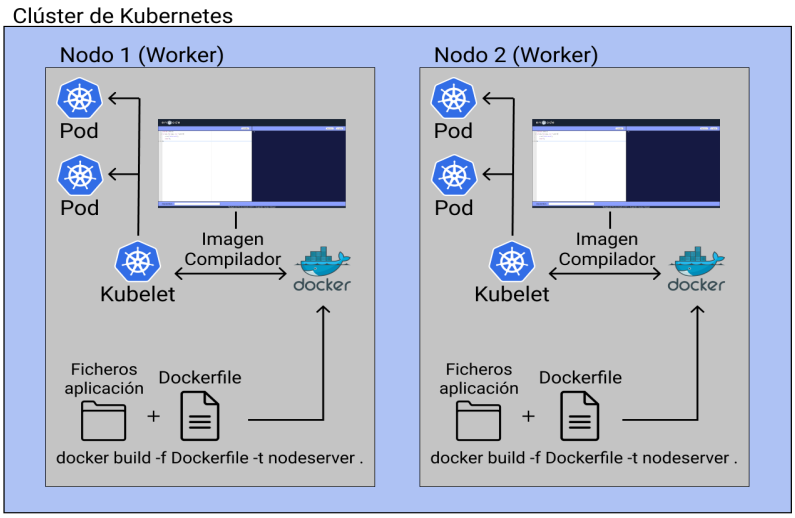

Figura 3.6: Relación entre *Kubernetes* y *Docker* en los nodos de trabajo.

Para terminar, se llevará a cabo la instalación e integración del sistema de monitorización *Prometheus* y el software de visualización *Grafana*, que posee una total integración con el anterior. Estas dos tecnologías permitirán tomar datos de consumo de recursos, uso de red, estimaciones de uso de la plataforma, etc, y pueden incluso soportar un sistema de alertas que se podría diseñar en un futuro. En la figura 3.7, se pueden ver de forma muy simplificada los flujos de datos que se producen mediante este sistema de monitorización.

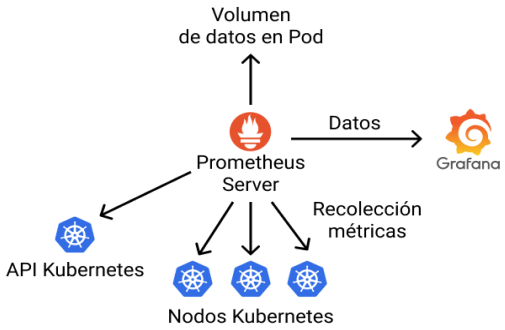

Figura 3.7: Diagrama simplificado de funcionamiento de *Prometheus-Grafana*

4

Implementación

En este capítulo, se presentará la implementación llevada a cabo a partir del diseño propuesto en el capítulo 3. A tal efecto, se dividirá el capítulo en dos secciones principales. La primera sección, tratará la solución que engloba el desarrollo de la aplicación a ejecutar en los contenedores y la generación de la imagen de *Docker*. La segunda parte, tratará las acciones necesarias para crear la infraestructura que constituirá el *clúster* de *Kubernetes*. Se explicarán en profundidad las configuraciones asociadas a todos los objetos de *Kubernetes* creados, y a su vez, se clasificarán dichos objetos en subsecciones, que harán referencia a la finalidad que desempeñan a la hora de proporcionar el servicio.

4.1. Aplicación del contenedor

El proceso de implementación de la aplicación a ejecutar en los contenedores se divide en dos partes, el desarrollo del servidor web, el cual comprende toda la lógica de la aplicación; y la composición de la imagen de *Docker*, a través de un archivo *Dockerfile*. A continuación, se analizará el proceso llevado a cabo.

4.1.1. Servidor web

La premisa sobre la que se parte para este desarrollo, es la explicada en la subsección 3.2.1. De esta forma, el primer paso a tomar fue la creación de un servidor web en *Node.js*, con la ayuda del módulo *Express*, y el establecimiento de una comunicación bidireccional y persistente entre el usuario y el servidor, mediante *Socket.IO*. Para ello, se crea el fichero *"server.js"*, como archivo principal, y la clase *"socketService.js"*, para gestionar los eventos asociados a la API de *Socket.IO*.

En el listado 4.1, se pueden ver las líneas del archivo *"server.js"* en las que se crea el servidor. Previamente al código del listado 4.1, se utiliza la función *use()* de *Express* para montar funciones *middleware*. Las funciones creadas tienen como objetivo especificar en qué directorio buscar los contenidos estáticos a la hora de responder peticiones, y habilitar *CORS*, para permitir obtener recursos desde cualquier dominio. Posteriormente, en la línea 1, se crea un servidor *HTTP*, al que se le asocia

la instancia de *Express*, para que éste funcione como *middleware*. Este paso es indispensable para hacer funcionar *Socket.IO*, ya que éste, sólo acepta servidores *HTTP* o *HTTPS*, no siendo compatible con instancias de módulos como *Express*. Por último, en las líneas 4 y 5, el servidor comienza a escuchar peticiones en el puerto 3000, y se asocia el servidor *HTTP* a la instancia de *Socket.IO*. En el apéndice A.1, se puede ver el código completo de este fichero.

Código 4.1: Fragmento del script que pone en funcionamiento el servidor *HTTP* de *Node.js* permitiendo conexiones bidireccionales mediante *Socket.IO*.

```
1 var server = require('http').Server(app);
2 var socketService = new SocketService();
3
4 server.listen(PORT, HOST); // PORT = 3000 | HOST = '0.0.0.0'
5 socketService.BindServer(server);
```

El siguiente paso, es configurar los escuchadores de eventos de *Socket.IO* para controlar las posibles interacciones entre el cliente y el servidor. Como todas las interacciones del usuario tienen como objetivo generar un resultado en una terminal creada en el servidor, previamente, se procederá a analizar la clase *PTYservice*, en la cual se encuentran las distintas funciones que interactúan directamente con la terminal.

La clase *PTYservice*, utiliza el módulo *node-pty*, para crear y manejar la terminal asociada al usuario, y el módulo *child_process*, para ejecutar comandos que no son necesarios que el usuario vea. Las funciones que aporta esta clase a la hora de configurar los eventos del *socket* son:

- **CreatePtyProcess().** Crea una instancia de *pty*, generando el tipo de terminal especificada en los atributos de la clase, y establece el evento por el cual, se envían al usuario las salidas generadas en la terminal a través de un *socket*.
- **Write(data).** Introduce la cadena de texto de la variable *"data"* en la terminal.
- **Compile(code, args).** Compila el código dado en la variable *"code"* y lo ejecuta con los argumentos especificados en *"args"*.
- **CleanFilesCreated().** Elimina los ficheros creados durante la compilación de código por el usuario.
- **Kill().** Destruye la terminal.

Para más detalles, el código de esta clase se encuentra en el apéndice A.2.

Como se introducía anteriormente, y conociendo las funciones procedentes de la clase *PTYservice*, que se utilizarán en las funciones siguientes, los distintos escuchadores de eventos que se han implementado en la clase *SocketService* son:

- **Evento "connection".** Se ejecuta cuando el usuario se conecta al *socket*. Se encarga de crear una instancia de la clase *PTYservice*, que es asociada a dicha conexión por el id del *socket*.
- **Evento "stdin".** Recoge una cadena de texto y la utiliza como argumento en la función *Write* de la clase *PTYservice*.

- **Evento “compile”.** Recoge dos cadenas de texto, una con el código y la otra con los argumentos de ejecución. Éstas serán pasadas como argumentos en la función *Compile* de la clase *PTYservice*.
- **Evento “reset”.** Destruye la terminal asociada a la conexión y crea una nueva instancia de *PTYservice*.
- **Eventos “disconnect” y “close”.** Destruye la terminal asociada a la conexión y elimina los ficheros de compilación generados por el usuario mediante la función *CleanFilesCreated* de la clase *PTYservice*.

La clase completa se puede consultar en el apéndice A.3.

Hasta ahora, sólo se ha presentado el código a ejecutar en el lado del servidor. En el lado del cliente, se han de tener en cuenta varios desarrollos. Primero, el desarrollo estético de la aplicación mediante *html* y el preprocesador *css*, *sass*, disponibles en el apéndice A.4 y en el apéndice A.5. Y segundo, el desarrollo de funciones asociadas a los módulos *Xterm*, *Socket.IO* y *Ace Editor*.

Por un lado, como se puede ver en la figura 4.1, se ha realizado una interfaz en forma de cuadrícula mediante una *grid-template*, y se ha creado una estructura de bloques diferenciados en la que integrar el editor de código, la representación de la terminal, un apartado de texto y algunos botones.

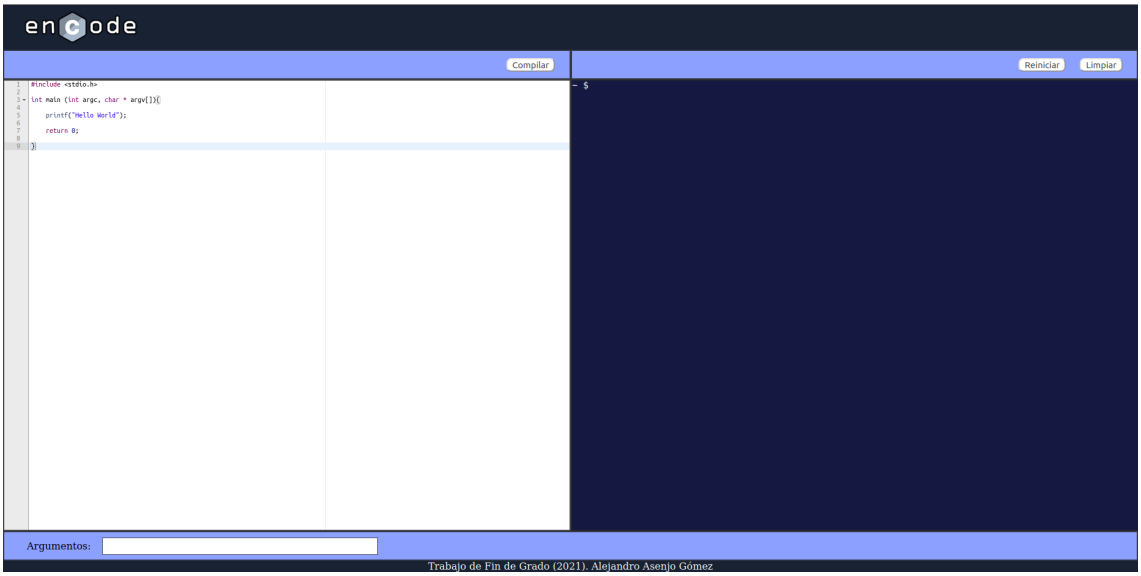

Figura 4.1: Interfaz web del compilador en línea.

Las acciones que puede llevar a cabo el usuario desde esta interfaz son:

- **Interacciones con la terminal.** La terminal del navegador recoge lo que el usuario teclea. Le permite borrar el texto actual mediante la tecla “Retroceso”, o copiar y pegar una cadena de texto. Además, al pulsar la tecla “Intro” desde la entrada de la terminal, dicho texto es enviado al servidor produciendo el evento “stdin” en el *socket* de comunicación.
- **Compilar y ejecutar el código del editor.** Mediante el botón “Compilar”, situado encima del editor de código, el texto escrito en dicho editor y los argumentos especificados en el apartado de texto de la zona inferior de la interfaz, son enviados al servidor produciendo el evento “compile” en el *socket* de comunicación.
- **Reiniciar la terminal.** Mediante el botón “Reiniciar”, se produce el evento “reset” en el *socket* de comunicación, ocasionando la terminación de los procesos asociados a la terminal y la sustitución de ésta por una nueva.
- **Limpiar la terminal.** Mediante el botón “Limpiar”, se borra todo el texto de la terminal del navegador.

En el fichero *ptyController*, disponible en el apéndice A.6, se implementan las funciones que controlan los eventos anteriores.

4.1.2. Docker

Docker es el software que se utiliza para ejecutar cada uno de los contenedores en los nodos de trabajo. De esta forma, el contenido del fichero *Dockerfile* marcará la configuración de la imagen de *Docker* a ejecutar en el contenedor. En el listado 4.2, se extraen los comandos más significativos a la hora de construir la imagen, aunque en el apéndice B se puede consultar el fichero completo.

Código 4.2: Extracto de algunos comandos del fichero *Dockerfile* de la imagen del compilador.

```
1 FROM alpine:3.13
2 ENV NODE_VERSION 15.14.0
3 RUN addgroup --gid 5000 compilerGroup && adduser -D compiler -h /home/compiler -u 5000 -G
      compilerGroup -s /bin/sh -g compiler
4 RUN apk add gcc libc-dev
5 COPY . .
6 RUN npm install
7 USER compiler
8 CMD ["node", "/srv/node/app/src/server.js"]
```

Para empezar, como se puede ver en las dos primeras líneas, se utiliza como base la imagen correspondiente a *Alpine* versión 3.13, con *Node.js* instalado, salvo por la modificación a la hora de crear el usuario que se usará al ejecutar el contenedor. Una vez instalada dicha base, se instala *gcc* y *libc-dev* para permitir la compilación y ejecución de código en C. Por otra parte, mediante el comando *COPY*, se copian todos los ficheros necesarios para la ejecución del servidor web en el contenedor, y se instalan los paquetes de *Node* necesarios mediante la herramienta *npm*. Para terminar, en las líneas 7 y 8, se establece el uso del usuario no *root* creado, y se configura la puesta en marcha del servidor para el momento en el que se ejecute el contenedor.

4.2. Clúster de Kubernetes

El proceso de implementación del *clúster* de *Kubernetes* tiene dos fases, una primera en la que se construye la base del *clúster*, asociada a instalar software, iniciar y configurar *Kubernetes* y enlazar componentes, entre otros; y una segunda en la que, de manera general, se configuran los objetos de *Kubernetes*.

4.2.1. Creación del clúster

En primer lugar, se crean dos máquinas virtuales *Linux*, una actuará como plano de control y la otra como nodo de trabajo. El proceso a seguir a continuación sería idéntico si se deseara aumentar el número de nodos de trabajo. Para la creación de las máquinas virtuales se ha utilizado *VMware Workstation*. Antes de nada, se deben configurar las distintas máquinas a nivel de red en modo *Bridge*. Este modo es imprescindible para que los nodos aparezcan como ordenadores independientes en la red y puedan interconectarse. Una vez creadas las máquinas virtuales, se pasará a seguir los siguientes pasos para establecer las configuraciones necesarias e instalar el *software* a utilizar en cada nodo:

- **Desactivar *Swap*.** *Kubernetes* requiere desactivar la memoria de intercambio de *Linux*, principalmente, para que la herramienta *Kubelet* no tenga problemas a la hora de gestionar la memoria de los *pods*.
- **Configurar la red.** Será necesario establecer una dirección ip estática para poder llevar a cabo las conexiones entre nodos.
- **Ejecución de contenedores.** Se instalará *Docker* como software responsable de ejecutar los contenedores y mantener las imágenes.
- **Instalar *Kubernetes*.** A partir del repositorio de *Kubernetes*, se instalará *Kubectl*, para desplegar y gestionar las aplicaciones; *Kubelet*, para asegurar la correcta ejecución de los contenedores; y *Kubeadm*, que será la herramienta que lleve a cabo las acciones para crear el *clúster*.

Una vez realizados estos pasos, se utiliza la herramienta *Kubeadm* para inicializar el nodo plano de control del *clúster*. Para ello, se hace uso del comando *init*. En el comando, se debe especificar la ip del nodo que actúa como plano de control, para que los nodos de trabajo se unan a dicha ip; y especificar el rango de direcciones ip que utilizará la red de los *pods*, lo que resulta en un bloque *CIDR* que es necesario para el funcionamiento de *Calico*. *Calico* actúa como *plugin* de red, permitiendo la conexión entre *pods*. Para las necesidades actuales del sistema, basta con que sea desplegado a partir de la configuración y recursos propuestos en la documentación oficial de *Calico* [28]. En el listado 4.3, se pueden ver los comandos utilizados para este fin. Además, en los últimos tres comandos, se especifica cómo acceder al *clúster* con un usuario no administrador. Esto consiste en hacer una copia del fichero "/kubernetes/admin.conf" en la ruta "$HOME/.kube/config", y hacer al usuario propietario de éste. Este fichero es usado para configurar el acceso a *Kubernetes* a la hora de usar *Kubectl*, y éste último, comprueba por defecto, la ruta "$HOME/.kube/config" en busca de esta configuración.

Código 4.3: Comandos que inicializan el nodo plano de control del *clúster* y configuran el uso de *Kubectl* para usuarios no *root*.

```
1 kubeadm init --pod-network-cidr=192.168.0.0/16 --apiserver-advertise-address=192.168.1.40
2
3 mkdir -p $HOME/.kube
4 sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
5 sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

El último paso a tomar, consiste en inicializar y asociar cada nodo trabajador al nodo plano de control. Para ello, se utilizará el comando *join* de *Kubeadm*. Este comando, requerirá únicamente la dirección ip del nodo que actúa como plano de control, el puerto asociado al plano de control dentro del nodo, un *token* y la clave pública del certificado de autoridad presentado por el plano de control. Estos datos serán obtenidos mediante *Kubeadm*.

Código 4.4: Comando que inicializa un nodo trabajador y lo asocia al nodo plano de control mediante *Kubeadm*.

```
kubeadm join 192.168.1.40:6443 --token ttps8x.6f4t2dytffb2nxzc --discovery-token-ca-cert-hash
    sha256:af6312919266b6e426408b30024fd075e1a4fc1826922f018cfb2f1dba142a11
```

4.2.2. Configuración del clúster

Tras completar el proceso de creación de los componentes del *clúster*, el siguiente paso es configurar los distintos objetos de *Kubernetes* que permitirán desplegar la aplicación y especificar las condiciones en las que será ejecutada. Para este fin, utilizaremos la herramienta *Kubectl*, que como ya se dijo, permite controlar el *clúster*. La exposición de la configuración se dividirá en varias categorías determinadas por funciones u objetivos. Cabe destacar que los posibles diagramas que se puedan encontrar en las siguientes secciones hacen referencia a la implementación actual sobre un nodo de trabajo. Si se instalaran más nodos de trabajo, algunas configuraciones podrían sufrir modificaciones superficiales, pero la creación de objetos y recursos de *Kubernetes* son aplicadas al *clúster* en su conjunto. Esto implica que los recursos se aplican a todos los nodos del *clúster*, y que las creaciones de *pods* se gestionan entre todos los nodos de trabajo existentes. Si no se quisiera que los *pods* fueran creados en cualquier nodo de trabajo, existen políticas de afinidad asociables a los distintos objetos de *Kubernetes*, en los que se puede especificar la elección de los nodos para la creación de los *pods*, según el cumplimiento de distintos requisitos.

Despliegue de la aplicación

El despliegue de la aplicación en el *clúster* será el primer paso a tomar. Éste irá evolucionando gradualmente a medida que se incluyan las configuraciones tratadas en las siguientes categorías.

Para empezar, se crea un espacio de nombres que funciona como un *clúster* virtual dentro del *clúster* físico. Esto se hace con la intención de separar y encapsular los recursos asociados a la aplicación, y facilitar la gestión de éstos. Creado éste, se crea un objeto de tipo *Deployment*, el cual se encargará de crear y replicar los *pods* que ejecutarán el contenedor con la imagen especificada. La configuración de este objeto permite configurar los *pods* y el *ReplicaSet* a desplegar, de forma que se simplifica la gestión de éstos. De esta manera, la configuración tiene dos objetivos, el primero, indicar

las condiciones de los *pods* a crear, y especificar al *ReplicaSet* qué *pods* debe gestionar y cuantas réplicas debe crear y mantener. En el listado 4.5, se puede ver el número de réplicas a crear de los *pods* que coinciden con la condición del *selector*, los cuales siguen la plantilla especificada en el apartado *template*.

Código 4.5: En este fragmento de la configuración del *Deployment* de *Kubernetes* se configura el número de replicas a crear del *pod* que posee el *label "app:server"*. Este *pod* ejecutará un contenedor con la imagen local que posee la etiqueta *"nodeserver:latest"*.

```
1  replicas: 2
2  selector:
3    matchLabels:
4      app: server
5  template:
6    metadata:
7      labels:
8        app: server
9        networkpolicy: denegar-trafico-saliente
10   spec:
11     containers:
12     -name: server
13       image: nodeserver:latest
14       imagePullPolicy: Never
15       ports:
16       -containerPort: 3000
```

Además, en la configuración asociada al *pod*, es posible limitar los recursos disponibles en el contenedor creado. Como se puede ver en el listado 4.6, se puede llevar a cabo una reserva inicial de recursos en el apartado *request*, que pueden ser sobrepasados hasta los límites especificados en el apartado *limits*. En caso de sobrepasar el límite de recursos, el contenedor sería reiniciado; pero el hecho de sobrepasar los recursos reservados, también abre la posibilidad de que el contenedor sea destruido si el nodo no tiene suficiente memoria.

Código 4.6: Configura las peticiones iniciales y los límites de recursos para cada contenedor.

```
1          requests:
2            memory: "256Mi"
3            cpu: "250m"
4            ephemeral-storage: "1Gi"
5          limits:
6            memory: "500Mi"
7            cpu: "500m"
8            ephemeral-storage: "2Gi"
```

Asimismo, cabe la posibilidad de configurar las condiciones necesarias para que la aplicación pueda recibir tráfico, mediante el campo *readinessProbes*, o comprobar si el contenedor necesita reiniciarse, con el campo *livenessProbe*. En el listado 4.7, se especifica en las dos pruebas que se realice una

conexión abriendo un *socket* en el puerto 3000 del contenedor, cada 3 segundos. Así, si el servidor *http* que escucha en ese puerto no está funcionando, la prueba fallará.

Código 4.7: Configuración de las pruebas de estado de los contenedores creados.

```
readinessProbe:
  tcpSocket:
    port: 3000
  initialDelaySeconds: 5
  periodSeconds: 3
  failureThreshold: 5
livenessProbe:
  tcpSocket:
    port: 3000
  initialDelaySeconds: 20
  periodSeconds: 3
  failureThreshold: 3
```

Por otra parte, se pueden restringir los permisos que tiene un contenedor a la hora de ejecutarse. Es una buena práctica eliminar privilegios y restringir el uso del usuario *root*, ya que si la imagen fuera importada de una entidad externa y utilizara privilegios de *root* por un mal diseño, o con intenciones dañinas, el contenedor no sería ejecutado al no cumplirse los requisitos especificados. En el caso actual, en el que la imagen es diseñada personalmente, esta configuración supone un complemento redundante frente a la configuración en el *Dockerfile*. En el listado 4.8, se restringen los privilegios de ejecución y se obliga a que el contenedor sea ejecutado por el usuario con id 5000, nunca el usuario *root*.

Código 4.8: Configuración del contexto de seguridad de los contenedores creados.

```
privileged: false
allowPrivilegeEscalation: false
runAsUser: 5000
runAsGroup: 5000
runAsNonRoot: true
capabilities:
  drop:
    -all
```

Una vez realizado el despliegue de la aplicación, se configurará el escalado horizontal de ésta a partir del recurso *Horizontal Pod Autoscaler*. Este recurso estará asociado al objeto *Deployment* anterior, y se encargará de escalar hacia arriba o hacia abajo, el número de *pods* replicados asociados al *Deployment*. Para ajustar el número de réplicas creadas, se especificará en su configuración el porcentaje de utilización de procesador y memoria, en el cual se produciría la creación de una nueva réplica. Además, se indica el número de réplicas mínimas y máximas que pueden existir para evitar situaciones en las que el servicio pierda disponibilidad, o se consuman excesivos recursos.

En la figura 4.2, se presenta, de forma esquemática, la jerarquía y relaciones que se producen al llevarse a cabo los anteriores despliegues.

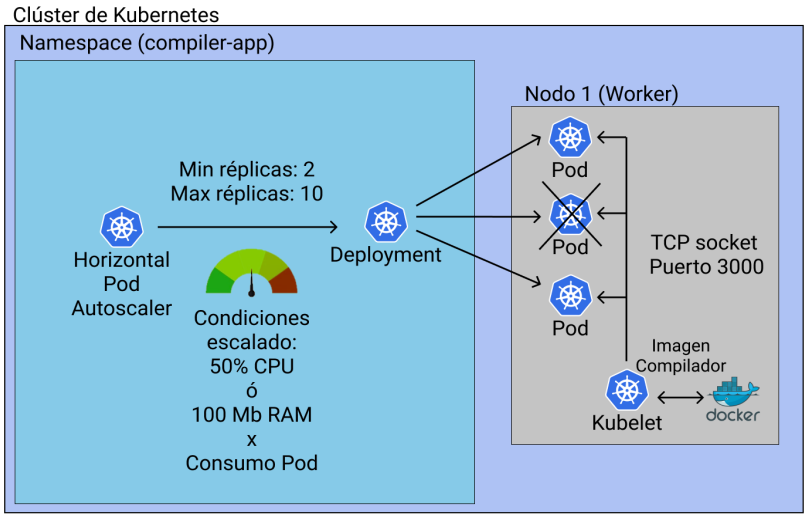

Figura 4.2: Diagrama de despliegue de la aplicación en *Kubernetes* que muestra las distintas jerarquías y relaciones entre los distintos recursos.

Exposición de la aplicación al usuario

La exposición del compilador en línea al usuario supone un enfoque muy diferente en el entorno actual, *On-premise*, frente a un entorno *Cloud*. En un entorno *Cloud*, el balanceador de carga a nivel de red está administrado por el proveedor *Cloud*, y simplifica la configuración enormemente a la hora de proveer acceso a los servicios. En el caso actual, el *clúster* no es accesible por usuarios externos y a nivel de seguridad, exponer el *clúster* directamente en la red pública es una solución que conlleva riesgos. De esta forma, la mejor solución es instalar un *proxy* inverso con *HAProxy*, como se hacia referencia en la sección 3.2.2.

Previamente a este paso, es necesario exponer los *pods* que ejecutan la aplicación mediante varios recursos a crear en *Kubernetes*. Para empezar, se crea un servicio de *Kubernetes* que tendrá por objetivo definir el conjunto de *pods* que ejecuta la aplicación y constituir un solo punto de acceso para todos ellos. Así, aunque se destruyan y creen nuevos *pods*, éstos serán siempre accesibles desde el servicio, que se encargará de asociarlos automáticamente. En concreto, el tipo de servicio creado es *ClusterIP*, el cual expone el servicio en una dirección ip interna del *clúster*, no accesible desde fuera de éste. Con esta solución, se posibilita acceder a la aplicación a partir de una dirección ip y un puerto único, desde dentro del *clúster*.

El siguiente paso, es hacer accesible el servicio de *Kubernetes* desde fuera del *clúster* a través de un objeto *Ingress*. Además, éste nos permitirá configurar el acceso al servicio a través de nombres de dominio y establecer un sistema de sesiones mediante el uso de *cookies*. En el listado 4.9, se pueden

ver las reglas que definen el *host*, la ruta mediante la cual acceder al servicio, y las anotaciones necesarias para establecer el sistema de sesiones.

Código 4.9: Reglas de redireccionamiento y sistema de sesiones del objeto *Ingress*.

```
  annotations:
    nginx.ingress.kubernetes.io/affinity: "cookie"
    nginx.ingress.kubernetes.io/session-cookie-name: "compiler"
    nginx.ingress.kubernetes.io/session-cookie-expires: "86400"
    nginx.ingress.kubernetes.io/session-cookie-max-age: "86400"
spec:
  rules:
  -host: compiler.example.com
    http:
      paths:
      -path: /
        pathType: Prefix
        backend:
          service:
            name: server-service
            port:
              number: 3000
```

Sin embargo, para que este recurso tenga efecto, es necesario desplegar un *Ingress Controller* en todos los nodos de trabajo del *clúster*, y un servicio que permita acceder a éste, desde fuera del *clúster*. De esta forma, como se puede ver en la figura 4.3, el *Ingress Controller* se desplegará en un *pod*, en cada uno de los nodos de trabajo. El servicio de tipo *NodePort* asociado a éste, permitirá acceder al *Ingress Controller* desplegado en cualquiera de los nodos. Este acceso, se hará a través de cualquiera de las direcciones ip de los nodos del *clúster*, y un puerto en específico. El puerto será especificado al desplegar el servicio, que en este caso será el 32239.

Para el despliegue del *Ingress Controller*, se ha utilizado, con pequeños cambios, la configuración recomendada en el repositorio de *Github* de *Kubernetes* para la instalación de la opción que utiliza *NGINX* como *proxy* [29]. También existen otras opciones posibles como *Traefik* o *HAProxy*, entre otros muchos.

Por último, como se muestra en la figura 4.3, y se explicaba al principio de la sección, se ha configurado un *proxy* inverso mediante *HAProxy*. Para lograr este objetivo, se ha creado una nueva máquina virtual con las siguientes características:

- **Sistema operativo *Alpine Linux*.** Esta distribución de *Linux* supone una buena elección a la hora de mejorar la seguridad y el consumo de recursos de la máquina.
- **Configuración de red.** En orden de permitir a usuarios acceder la máquina, y ésta pueda acceder a los nodos del *clúster*, se ha configurado la red en modo *bridge* y se le ha establecido una dirección ip estática.
- **Instalación de paquetes.** Se ha instalado el paquete *HAProxy* mediante la herramienta *apk*.

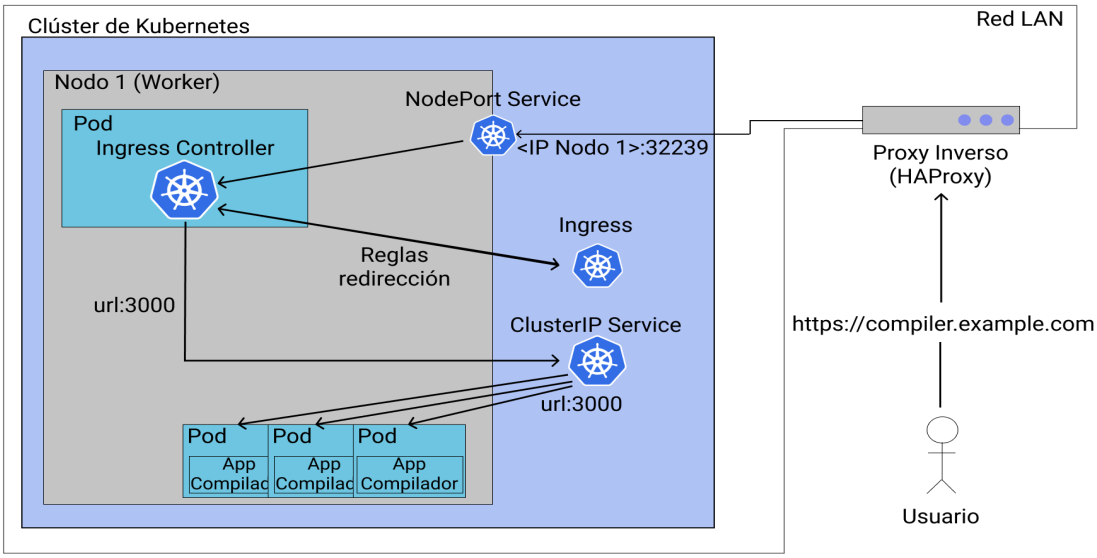

Figura 4.3: Estructura para la redirección del tráfico de red a la aplicación.

Sin entrar en detalles, la configuración de *HAProxy* redirigirá todas las peticiones que reciba en el puerto 443, a los nodos de trabajo del *clúster* de *Kubernetes*, e implementará un sistema de sesiones por *cookies*.

Código 4.10: Configuración de *HAProxy* como *proxy* inverso.

```
frontend main
    bind *:443 ssl crt /etc/ssl/certs/ssl.pem
    default_backend  compiler

backend compiler
    balance roundrobin
    option tcp-check
    cookie server insert indirect nocache
    server compiler1 192.168.1.42:32239 check port 32239 cookie compiler1 ssl verify none
```

Políticas de red

Otra de las medidas de seguridad que se han tomado para proteger el sistema, es configurar la política de red de los *pods* en los que se ejecutará la aplicación. De esta forma, se ha configurado una política de red que restringe todo el tráfico saliente procedente de los *pods*. Esto tiene por objetivo evitar que los usuarios puedan realizar acciones dentro del contenedor que impliquen conectarse a la red externa al *clúster*, o a recursos dentro del *clúster*. Es importante destacar que las políticas de red sólo pueden ser implementadas por el *plugin* de red instalado en el *clúster*, por tanto, sin la instalación de *Calico*, esta característica no estaría disponible. La configuración del recurso de *Kubernetes*,

NetworkPolicy, que implementa lo anterior, se encuentra disponible en el apéndice C.3.

Tareas automáticas de mantenimiento

Como se declara en la sección 3.1.1 y la sección 3.1.2, la aplicación almacenará archivos creados por el usuario, fruto de la compilación de código, o creados manualmente mediante el intérprete de comandos. Además, se pueden generar procesos con largos tiempos de ejecución, o que hayan sufrido errores y no hayan sido finalizados por los usuarios. Por tales razones, se implementarán tareas programadas que serán ejecutadas por *pods*. Esta implementación consiste en la creación de un objeto *CronJob* que automatizará la creación de un objeto *job* para cada ejecución de la tarea. Así mismo, el objeto *job* se encargará de crear *pods* que ejecutarán la acción hasta que ésta sea completada correctamente, o se produzcan 6 errores seguidos y se considere la tarea como fallida.

De esta forma, una de las tareas programadas consistirá en eliminar todos los archivos que fueron modificados hace más de un minuto, y que hayan sido generados por el usuario en las carpetas *"compiler"* y *"tmp"*, que son las únicas en las que posee permiso de escritura. Por otro lado, en la misma tarea, se terminarán todos los procesos creados por el usuario con un tiempo de ejecución mayor a 45 segundos. Esta tarea, se ejecutará cada minuto y hará uso de la herramienta *Kubectl* para permitir ejecutar comandos en los *pods* que ejecutan la aplicación.

Código 4.11: Comando a ejecutar en la tarea de mantenimiento que elimina los archivos creados por el usuario.

```
for pod in $(kubectl get pods | awk '/^server-deployment/ {print $1}');
    do kubectl exec $pod --/bin/sh -c "find /home/compiler -mindepth 1 -mmin +1 -delete && find /tmp -
        mindepth 1 -mmin +1 -user 5000 -delete
    && (ps -eo pid,user,etime | sed '1d' | awk '{if(\$1!=1 && \$2==\"compiler\" && (split(\$3,a,\":\")
        && (a[1]>0 || a[2]>=45))) print \$1}' | sort -r -n | xargs -r kill -9)"; done;
```

La otra tarea programada, reiniciará los *pods* que ejecutan la aplicación una vez cada 12 horas. Para ello, un *pod* lanzará una instrucción mediante *Kubectl* en la que ordene realizar un reinicio del *Deployment* que administra los *pods* que ejecutan la aplicación. En ningún momento el servicio dejará de estar disponible, ya que primero, se crearán al menos dos nuevos *pods*, y hasta que no estén en funcionamiento, no se comenzará a sustituir los *pods* antiguos.

Código 4.12: Comando a ejecutar en la tarea de mantenimiento que reinicia los *pods* que ejecutan la aplicación.

```
kubectl rollout restart deployment/server-deployment
```

Monitorización del clúster

La monitorización tendrá como objetivo controlar el uso de recursos del *clúster* y examinar el tráfico de red que se produce en éste. Para este fin, utilizaremos la herramienta de monitorización *Prometheus*, la cual ofrece una gran integración con *Kubernetes*. Además, los componentes de *Kubernetes*, publican sus métricas en el formato que utiliza *Prometheus* a partir de una ruta del servidor *http*.

Para el caso de los recursos del *clúster*, se utilizarán las métricas que proporciona el componente *Kubelet*, instalado en todos los nodos. *Kubelet* aportará las métricas de los contenedores que se encuentran en ejecución en cada uno de los nodos, de forma que, una vez organizada la información, se puede conocer el estado de todos los nodos del *clúster*. Estas métricas serán obtenidas a partir del agente *cAdvisor*, integrado en *Kubelet*, que específicamente monitoriza los contenedores en ejecución y tiene soporte nativo para *Docker*. La ruta para obtener las métricas del *cAdvisor* es *"/metrics/cadvisor"*, quedando la configuración de *Prometheus* de la siguiente forma.

Código 4.13: Fragmento de la configuración de *Prometheus* que permite obtener datos del agente *cAdvisor*.

```
-job_name: 'kubernetes-cadvisor'
  scheme: https
  metrics_path: /metrics/cadvisor
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    insecure_skip_verify: true
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  -role: node
```

Por otro lado, para monitorizar el tráfico de red, se obtendrán las métricas que exporta el *Ingress* de *NGINX* que se instaló para exponer el servicio. Para ello, se especificará el puerto 10254, como el puerto del que *Prometheus* debe exportar las métricas. Esto se realiza mediante anotaciones en los objetos *Ingress* e *Ingress Controller*, que se pueden ver en el listado 4.14.

Código 4.14: Anotaciones en *Ingress* e *Ingress Controller* para permitir a *Prometheus* obtener las métricas que exporta el objeto *Ingress Controller*.

```
annotations:
    prometheus.io/port: "10254"
    prometheus.io/scrape: "true"
```

En el listado 4.15, se puede ver la configuración de *Prometheus* que especifica la obtención de métricas del *Ingress Controller*.

Código 4.15: Fragmento de la configuración de *Prometheus* que permite obtener datos del *Ingress Controller* de *NGINX*.

```
-job_name: 'ingress-nginx-endpoints'
  kubernetes_sd_configs:
  -role: pod
    namespaces:
      names:
      -ingress-nginx
```

El código utilizado como base para el despliegue y configuración de *Prometheus* es el aportado en el repositorio de *Kubernetes* [29]. A partir de esta base, ha sido necesario realizar varios cambios importantes. Por un lado, se ha cambiado completamente la configuración de roles de privilegios para permitir acceder a las métricas aportadas por *Kubelet* en los distintos nodos. Por otro lado, se ha tenido que indicar en la configuración de *Prometheus* cómo descubrir las métricas de *Kubelet*, y de qué manera tratarlas.

Una vez desplegado *Prometheus* con dichas configuraciones, se procederá a desplegar *Grafana* para visualizar los datos obtenidos por el primero. Los tableros que visualizarán los datos, serán importados de la página oficial de *Grafana* [30]. El tablero orientado a mostrar los datos obtenidos del *Ingress*, será el compatible con la configuración aportada en el repositorio de *Kubernetes* [29] [31]. Por otro lado, el tablero que mostrará los recursos del *clúster*, será obtenido a través del repositorio oficial de *Grafana* [32]. Dicho tablero, expone de forma muy clara y amplia la situación de los recursos gracias a la categorización que hace de los datos. En el apéndice C.1 y en el apéndice C.2, respectivamente, se muestran los distintos gráficos aportados en los dos tableros.

Para el despliegue de *Grafana*, se ha utilizado la configuración aportada en el repositorio de *Kubernetes* [29]. Los servicios de *Prometheus* y *Grafana* serán accesibles desde la dirección ip de cualquier nodo del *clúster*, incluido el plano de control, en el puerto especificado al crear el servicio de tipo *NodePort*, para cada uno de los despliegues.

5 Pruebas

En este capítulo, se probará la implementación que se ha llevado a cabo, de manera que se pueda ver el resultado final de la aplicación web, y se compruebe el efecto real de las configuraciones establecidas en el *clúster* de *Kubernetes*. Las pruebas documentadas a continuación, se han realizado sobre un sistema con un procesador Intel Core I5-7600K con una frecuencia de 3.80 GHz y 4 núcleos, y 16 GB de memoria RAM a 2400 MHz.

Respecto al *clúster* de *Kubernetes*, el nodo plano de control tiene a su disposición dos núcleos del procesador, 4 GB de RAM y 50 GB de espacio de disco duro. Por otro lado, el nodo trabajador tiene a su disposición dos núcleos del procesador, 5 GB de RAM y 50 GB de espacio de disco duro. Por último, la máquina virtual en la que se ejecuta el *proxy* inverso, puede utilizar un núcleo de procesador, un GB de RAM y 10 GB de disco duro, aunque su consumo es mínimo.

5.1. Acceso a la aplicación

En esta sección, se probará el correcto funcionamiento de la implementación expuesta en la subsección 4.2.2. Para empezar, se comienza poniendo en funcionamiento el nodo plano de control, sin iniciar el nodo de trabajo. De esta forma, el servicio de *Kubernetes*, que expone los *pods* en los que corre la aplicación, no tendrá ningún *pod* asociado. En la figura 5.1, se puede ver cómo el servicio no ha encontrado ningún *pod* al que redirigir el tráfico.

```
Rules:
  Host                Path  Backends
  ----                ----  --------
  compiler.example.com
                      /   server-service:3000 (<none>)
```

Figura 5.1: Reglas de redireccionamiento del *Ingress* sin *pods* disponibles.

Por otro lado, si se inicia el nodo de trabajo y por tanto, comienzan a ejecutarse los *pods* que ejecutan la aplicación, dichos *pods* son asociados al servicio. En la figura 5.2, se puede ver cómo el servicio ahora redirige el tráfico a esos *pods*.

```
Rules:
  Host                Path  Backends
  ----                ----  --------
  compiler.example.com
                      /   server-service:3000 (192.168.190.240:3000,192.168.190.251:3000)
```

Figura 5.2: Reglas de redireccionamiento del *Ingress* con *pods* disponibles.

5.2. Funcionalidad básica del compilador

En esta sección, se mostrará un ejemplo de uso general del compilador en línea, el cual fue presentado en la subsección 4.1.1. En la figura 5.3 y en la figura 5.4, se presenta una situación en la que el usuario compila y ejecuta un código con algunas advertencias. En el código, se piden por teclado varios números. Estos números son sumados y posteriormente, el resultado es multiplicado por el número introducido como segundo argumento de ejecución. Además, para probar correctamente la entrada de argumentos de ejecución, el primer argumento de ejecución introducido es impreso por pantalla en el último mensaje producido por el programa.

```c
#include <stdio.h>

int main (int argc, char * argv[]){

    int * numeros = malloc(5 * sizeof(int));
    int factor = atoi(argv[2]);
    int resultado = 0;

    printf("Introduce 5 números a sumar (separados por espacios):\n");
    scanf("%d %d %d %d %d", &numeros[0], &numeros[1], &numeros[2], &numeros[3], &numeros[4]);

    for(int i = 0; i < 5; i++){
        resultado += numeros[i];
    }

    resultado *= factor;

    printf("Resultado del cálculo con nombre %s es: %d\n", argv[1], resultado);

    return 0;
}
```

Figura 5.3: Código en C escrito en el editor del compilador en línea para ser ejecutado.

```
~ $ gcc -g3 -o3 /tmp/jUUByBKLKUKb4JaUAAAB.c -o /tmp/jUUByBKLKUKb4JaUAAAB.out

/tmp/jUUByBKLKUKb4JaUAAAB.c: In function 'main':

/tmp/jUUByBKLKUKb4JaUAAAB.c:5:18: warning: implicit declaration of function 'malloc'
[-Wimplicit-function-declaration]
    5 |   int * numeros = malloc(5 * sizeof(int));
      |                   ^~~~~~
/tmp/jUUByBKLKUKb4JaUAAAB.c:5:18: warning: incompatible implicit declaration of built
-in function 'malloc'
/tmp/jUUByBKLKUKb4JaUAAAB.c:2:1: note: include '<stdlib.h>' or provide a declaration
of 'malloc'
    1 | #include <stdio.h>
  +++ |+#include <stdlib.h>
    2 |
/tmp/jUUByBKLKUKb4JaUAAAB.c:6:15: warning: implicit declaration of function 'atoi' [-
Wimplicit-function-declaration]
    6 |   int factor = atoi(argv[2]);
      |                ^~~~

~ $ /tmp/jUUByBKLKUKb4JaUAAAB.out "Prueba
de argumentos" 5

Introduce 5 números a sumar (separados por espacios):


1 2 3 4 5

Resultado del cálculo con nombre Prueba de argumentos es: 75
```

Figura 5.4: Resultado de la compilación y ejecución del código en C, escrito en el editor del compilador en línea, que es mostrado en la terminal.

Como se puede ver en la figura 5.4, en la terminal se mostrarán los errores haciendo referencia a los números de línea correspondientes al editor de código del navegador, lo cual facilita al usuario la

comprensión y corrección de dichos errores.

5.3. Políticas de seguridad

En esta sección, se van a mostrar dos características de seguridad que se han configurado para los contenedores. Por un lado, si se trata de ejecutar en un contenedor de un *pod* la imagen del compilador en línea, o cualquier imagen, como usuario *root*, se produce en el *pod* el error mostrado en la figura 5.5. Para simular este error, se ha modificado la imagen del compilador, de forma que no se cree el usuario *compiler* y se utilice, por defecto, el usuario *root*.

```
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  2m4s               default-scheduler  Successfully assigned compiler-app/server-deployment-9667c7f6f-mkn5t to workernode
  Normal   Pulled     5s (x12 over 2m3s)  kubelet            Container image "nodeserver:latest" already present on machine
  Warning  Failed     5s (x12 over 2m3s)  kubelet            Error: container has runAsNonRoot and image will run as root (pod: "server-deployment
-9667c7f6f-mkn5t_compiler-app(924827da-e5a8-4fd1-94f9-774dcf6e3862)", container: server)
```

Figura 5.5: Error al intentar ejecutar una imagen como *root* en un contenedor.

Por otro lado, no es posible realizar conexiones a la red externa e interna al *clúster* desde los *pods* que ejecutan la imagen del compilador. Esto se puede probar utilizando el comando *ping*. Este comando no puede ser usado por el usuario *compiler*, el cual utilizaría el usuario al acceder a la aplicación. En la figura 5.6, se muestra el resultado de ejecutar el comando *ping* con el usuario *compiler*. Esto es debido a las configuraciones de seguridad, referentes a los contenedores, del objeto *deployment*. Para simular esta prueba, se ha modificado la configuración para ejecutar el contenedor con el usuario *root*. En la figura 5.7 y la figura 5.8, se muestra una conexión a otro *pod* dentro del mismo nodo de trabajo, y una conexión al servidor *DNS* de *Google*, 8.8.8.8. Las dos conexiones no tienen éxito debido a la política de red configurada.

```
ping 192.168.190.253

PING 192.168.190.253 (192.168.190.253): 56 data bytes
ping: permission denied (are you root?)
```

Figura 5.6: Error al ejecutar el comando *ping* como usuario no *root* desde la terminal del compilador en línea.

```
ping 192.168.190.200 -w 10

PING 192.168.190.200 (192.168.190.200): 56 data bytes


--- 192.168.190.200 ping statistics ---
11 packets transmitted, 0 packets received, 100% packet loss
```

Figura 5.7: Resultado fallido del intento de conexión de un *pod* a otro mediante el comando *ping*. Todos los paquetes transmitidos se han perdido.

```
ping 8.8.8.8 -w 10

PING 8.8.8.8 (8.8.8.8): 56 data bytes


--- 8.8.8.8 ping statistics ---
11 packets transmitted, 0 packets received, 100% packet loss
```

Figura 5.8: Resultado fallido del intento de conexión de un *pod* al servidor *DNS* de *Google* mediante el comando *ping*. Todos los paquetes transmitidos se han perdido.

5.4. Disponibilidad y mantenimiento de la aplicación

Para esta sección, se va mostrar un ejemplo del funcionamiento de las pruebas de estado *Readiness* y *Liveness*, que se llevan a cabo sobre los *pods* que ejecutan la aplicación del compilador. Para esta prueba, se va a provocar la situación en la que uno de los dos *pods* existentes descargue la imagen del compilador correcta, y el otro, descargue una imagen en la que el servidor escuche en un puerto diferente al 3000. En la figura 5.9, se pueden ver las condiciones de los *pods*.

```
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
```

(a) Estado del *pod* con la imagen correcta.

```
Conditions:
  Type              Status
  Initialized       True
  Ready             False
  ContainersReady   False
  PodScheduled      True
```

(b) Estado del *pod* con la imagen modificada.

Figura 5.9: Condiciones de los *pods* al ser ejecutados. El *Pod* que representa la figura 5.9(b) muestra que no está preparado al no cumplir las pruebas establecidas en *Readiness*, a diferencia del representado en la figura 5.9(a).

Esta situación es registrada en los eventos del *pod*, de la forma que se muestra en la figura 5.10. Cabe destacar que el contenedor se ha reiniciado, a consecuencia de la prueba de estado *Liveness*. Esto implica que se ha descargado la imagen otra vez por cada reinicio. Igualmente, como no se ha restaurado la imagen correcta en el nodo trabajador, siempre se descarga la imagen incorrecta produciendo otro error. Por otro lado, el *pod* dejaría de estar asociado al servicio al no encontrarse en las condiciones correctas.

```
Events:
  Type     Reason     Age                    From               Message
  ----     ------     ----                   ----               -------
  Normal   Scheduled  8m19s                  default-scheduler  Successfully assigned compiler-app/server-deployment-695fcbdbb9-8gvkq to workernode
  Normal   Pulled     8m19s                  kubelet            Container image "nodeserver:latest" already present on machine
  Normal   Created    8m19s                  kubelet            Created container server
  Normal   Started    8m19s                  kubelet            Started container server
  Warning  Unhealthy  7m53s (x3 over 7m59s)  kubelet            Liveness probe failed: dial tcp 192.168.190.199:3000: connect: connection refused
  Normal   Killing    7m53s                  kubelet            Container server failed liveness probe, will be restarted
  Warning  Unhealthy  3m17s (x96 over 8m14s) kubelet            Readiness probe failed: dial tcp 192.168.190.199:3000: connect: connection refused
```

Figura 5.10: Registro de eventos de un *pod* que no cumple las pruebas de *Readiness* y *Liveness*.

6

Conclusiones y trabajo futuro

6.1. Conclusiones

Una vez analizados la implementación y los resultados de ejecución obtenidos, se puede concluir que los objetivos establecidos en la sección 1.2 han sido satisfechos. Para recapitular, se ha desarrollado la aplicación web y se ha gestionado como una imagen de *Docker*. Por otro lado, se ha creado y configurado un *clúster* de *Kubernetes*, que logra proporcionar a la aplicación una alta disponibilidad, la hace resiliente a errores, y lo más importante, automatiza todo el proceso de despliegue, mantenimiento y exposición de la aplicación. Por último, se han proporcionado las herramientas *Prometheus* y *Grafana* para permitir monitorizar el uso del *clúster*.

Más allá de los objetivos técnicos logrados, en este trabajo, se presenta una plataforma que cumple el propósito principal de aportar un nuevo servicio a la plataforma de aprendizaje *ENCODE*. Esta plataforma ha sido desarrollada teniendo en cuenta el entorno de instalación de *ENCODE*, lo que permite una fácil instalación en éste, sin prácticamente modificaciones.

Así, a partir de este trabajo, se abre un amplio abanico de proyectos que mejoren y expandan la idea original del compilador en línea dentro de la plataforma *ENCODE*.

6.2. Trabajo futuro

Como la motivación principal de este trabajo es conseguir aportar un nuevo servicio a la plataforma de aprendizaje *ENCODE*, la primera tarea pendiente que se presenta es instalar la plataforma presentada en este trabajo en las instalaciones de la Escuela Politécnica Superior. Una vez instalada la plataforma, sería necesario hacerla accesible a través del dominio que utiliza *ENCODE*.

A partir del punto anterior, sería necesario diseñar y configurar la integración entre la plataforma de *ENCODE*, y el compilador en línea. Este punto no tiene porqué ser el siguiente paso a tomar, ya que el servicio de compilador en línea puede comenzar a ser usado como un servicio complementario, no propiamente integrado en la misma plataforma.

Por otro lado, se podría añadir una funcionalidad de depuración de programas sobre la aplicación de programación en C, creada en este trabajo. Sería una característica muy útil para los usuarios que implicaría rediseñar la aplicación web. Lo ideal sería añadir a la interfaz web una forma de interactuar con el depurador de la forma más gráfica posible.

Por último, si se integrara satisfactoriamente con la plataforma de *ENCODE*, una idea que surgió de forma recurrente al presentar este trabajo, era terminar adaptándolo para que funcionase como un sistema de comprobación de las soluciones que implementaban los usuarios durante las lecciones de los cursos de *ENCODE*. Esto implicaría una completa integración de la plataforma de este trabajo de fin de grado, con el desarrollo de *ENCODE*, por tanto es un futuro trabajo exigente y complejo, que previsiblemente sea necesario dividir en partes.

Bibliografía

[1] "Cncf survey 2020." https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf. Consultado: 2021-05-18.

[2] S. Moore, "Gartner forecasts strong revenue growth for global container management software and services through 2024." https://www.gartner.com/en/newsroom/press-releases/2020-06-25-gartner-forecasts-strong-revenue-growth-for-global-co. Consultado: 2021-05-11.

[3] A. Frías Díaz, "Encode: Tutor online de programación," bachelor's thesis, UAM. Departamento de Ingeniería Informática, 2019.

[4] A. Bonilla Trueba, "Encode: Tests online para el aprendizaje de c," bachelor's thesis, UAM. Departamento de Ingeniería Informática, 2020.

[5] E. Rico Mercader, "Encode: módulo de seguimiento de estudiantes," bachelor's thesis, UAM. Departamento de Ingeniería Informática, 2020.

[6] "How technology incumbents can realize their cloud growth ambitions." https://www.bain.com/insights/how-technology-incumbents-can-realize-their-cloud-\growth-ambitions/. Consultado: 2021-05-15.

[7] "The changing faces of the cloud." https://media.bain.com/Images/BAIN_BRIEF_The_Changing_Faces_of_the_Cloud.pdf. Consultado: 2021-05-15.

[8] "Gartner forecasts worldwide public cloud end-user spending to grow 18% in 2021." https://www.gartner.com/en/newsroom/press-releases/2020-11-17-gartner-forecasts-worldwide-public-cloud-end-user-\spending-to-grow-18-percent-in-2021. Consultado: 2021-05-15.

[9] "What is kubernetes?." https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Consultado: 2021-05-15.

[10] "chroot." https://en.wikipedia.org/wiki/Chroot. Consultado: 2021-05-15.

[11] P.-H. Kamp and R. Watson, "Building systems to be shared, securely: Want to securely partition vms? one option is to put 'em in jail.," *Queue*, vol. 2, p. 42–51, July 2004.

[12] "namespaces(7) - linux manual page." https://man7.org/linux/man-pages/man7/namespaces.7.html. Consultado: 2021-05-15.

[13] "cgroups(7) - linux manual page." https://man7.org/linux/man-pages/man7/cgroups.7.html. Consultado: 2021-05-15.

[14] B. Golub, "dotcloud, inc. is becoming docker, inc.." https://www.docker.com/blog/dotcloud-is-becoming-docker-inc/. Consultado: 2021-05-15.

[15] S. Hykes, "Docker 0.9: introducing execution drivers and libcontainer." https://www.docker.com/blog/docker-0-9-introducing-execution-drivers-and-libcontainer/.

Consultado: 2021-05-15.

[16] "Docker architecture." https://docs.docker.com/get-started/overview/#docker-architecture. Consultado: 2021-05-15.

[17] J. Kreisa, "Docker index: Dramatic growth in docker usage affirms the continued rising power of developers." https://www.docker.com/blog/docker-index-dramatic-growth-in-docker-usage-affirms-the-continued-\rising-power-of-developers/. Consultado: 2021-05-15.

[18] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," *Queue*, vol. 14, p. 70–93, Jan. 2016.

[19] "Introducción a la arquitectura de kubernetes." https://www.redhat.com/es/topics/containers/kubernetes-architecture. Consultado: 2021-05-16.

[20] "Kubernetes components." https://kubernetes.io/docs/concepts/overview/components/. Consultado: 2021-05-16.

[21] "Acerca de node.js." https://nodejs.org/es/about/. Consultado: 2021-05-22.

[22] "npm docs." https://docs.npmjs.com/. Consultado: 2021-05-22.

[23] "Express. fast, unopinionated, minimalist web framework for node.js." https://expressjs.com/. Consultado: 2021-05-23.

[24] "Socket.io." https://socket.io/docs/v4. Consultado: 2021-05-23.

[25] "Xterm.js." https://xtermjs.org/. Consultado: 2021-05-23.

[26] "Node-pty." https://www.npmjs.com/package/node-pty. Consultado: 2021-05-23.

[27] "Understanding kubernetes objects." https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. Consultado: 2021-05-24.

[28] "Install calico networking and network policy for on-premises deployments." https://docs.projectcalico.org/getting-started/kubernetes/self-managed-onprem/onpremises. Consultado: 2021-05-31.

[29] "Nginx ingress controller for kubernetes." https://github.com/kubernetes/ingress-nginx/. Consultado: 2021-06-01.

[30] "Dashboards. official and community built dashboards." https://grafana.com/grafana/dashboards. Consultado: 2021-06-03.

[31] "Dashboards. nginx ingress controller." https://grafana.com/grafana/dashboards/9614. Consultado: 2021-06-03.

[32] "Dashboards. kubernetes cluster monitoring (via prometheus)." https://grafana.com/grafana/dashboards/315. Consultado: 2021-06-03.

[33] "Official docker image for node.js." https://github.com/nodejs/docker-node. Consultado: 2021-06-11.

Apéndices

A

Recursos servidor web

En este apéndice, se presentan los distintos recursos utilizados en el desarrollo del servidor web. Se pretende que faciliten la comprensión de estos recursos a la hora de ser referenciados en el cuerpo del documento.

A.1. Script principal del servidor web

Código A.1: Script que pone en funcionamiento el servidor *Http* de *Node.js* permitiendo conexiones bidireccionales mediante *Socket.IO*.

```
'use strict';
const express = require('express');
const SocketService = require("./public/js/socketService");
// Constants
const PORT = 3000;
const HOST = '0.0.0.0';
// App
const app = express();
app.use(express.static(__dirname + '/public'));
app.use((req, res, next) => {
    res.header('Access-Control-Allow-Origin', '*');
    res.header('Access-Control-Allow-Headers', 'Authorization, X-API-KEY, Origin, X-Requested-With,
        Content-Type, Accept, Access-Control-Allow-Request-Method');
    res.header('Access-Control-Allow-Methods', 'GET, POST, OPTIONS');
    res.header('Allow', 'GET, POST, OPTIONS');
    next();
})
var server = require('http').Server(app);
var socketService = new SocketService();
server.listen(PORT, HOST);
socketService.BindServer(server);
```

A.2. Clase *PTYservice*

Código A.2: Clase *PTYservice* orientada a proporcionar funciones para crear y manejar la terminal que ejecuta el código del compilador.

```
var pty = require('node-pty');
const { exec } = require("child_process");

class PTYservice{
    constructor(socket) {
        this.shell = 'sh';
        this.ptyProcess = null;
        this.socket = socket;

        // Create instance of pty
        this.CreatePtyProcess();
    }

    // Create a pty instance with the shell specified in the constructor
    CreatePtyProcess() {
        this.ptyProcess = pty.spawn(this.shell, [], {
            name: 'xterm-color',
            cols: 85,
            rows: 42,
            cwd: process.env.HOME,  // Start path for the terminal
            env: process.env  // Environment variables
        });

        // Outputs generated in terminal are sent to the client socket to be displayed on the browser
        this.ptyProcess.on("data", (data) => {
            this.socket.emit("stdout", data);
        });
    }

    // Send user input to the Terminal Process stdin
    //  @param {*} data Input from user written on the browser UI
    Write(data) {
        this.ptyProcess.write(data + "\n");
    }
```

Código A.3: Clase *PTYservice* orientada a proporcionar funciones para crear y manejar la terminal que ejecuta el código del compilador.

```
    // Compile and execute a program given the code and execution arguments
    // @param {*} code Code to compile
    // @param {*} args Arguments of execution
    Compile(code, args) {
        var fileName = this.socket.id + ".c";
        var executableName = this.socket.id + ".out";

        // New console in order to not show this message to the user
        exec("echo '" + code + "'" + " > " + fileName + "\n", {cwd: "/tmp"});

        this.ptyProcess.write("gcc -g3 -o3 /tmp/" + fileName + " -o /tmp/" + executableName + "\n");

        this.ptyProcess.write("/tmp/" + executableName + " " + args + "\n");
    }

    // Delete files created from compilation
    CleanFilesCreated(){
        var fileName = this.socket.id + ".c";
        var executableName = this.socket.id + ".out";

        exec("rm " + fileName + "\n", {cwd: "/tmp"});
        exec("rm " + executableName + "\n", {cwd: "/tmp"});
    }

    // Kill the terminal process
    Kill() {
        this.ptyProcess.kill();
    }
}

module.exports = PTYservice;
```

A.3. Clase *SocketService*

Código A.4: Clase *SocketService* orientada a configurar los escuchadores de eventos de *Socket.IO* para controlar las posibles interacciones entre el cliente y el servidor.

```
const SocketIO = require("socket.io");
const PTYservice = require("./ptyService");
class SocketService {
    constructor() {
        this.terminales = {};
    }
    BindServer(server) {
        if(!server) {
            throw new Error("Occur an error with the server...");
        }
        const io = SocketIO(server);
        io.on("connection", (socket) => {
            this.terminales[socket.id] = new PTYservice(socket);
            socket.on("stdin", (stdin) => {
                this.terminales[socket.id].Write(stdin);
            });
            socket.on("compile", (data) => {
                this.terminales[socket.id].Kill();
                this.terminales[socket.id] = new PTYservice(socket);
                this.terminales[socket.id].Compile(data.code, data.args);
            });
            socket.on("reset", (message) => {
                this.terminales[socket.id].Kill();
                this.terminales[socket.id] = new PTYservice(socket);
            });
            socket.on("disconnect", () => {
                this.terminales[socket.id].CleanFilesCreated()
                this.terminales[socket.id].Kill();
            });
            socket.on("close", () => {
                this.terminales[socket.id].CleanFilesCreated()
                this.terminales[socket.id].Kill();
            });
        });
    }
}
module.exports = SocketService;
```

A.4. Documento *index.html*

Código A.5: Documento que constituye la estructura de la web del compilador.

```
<!doctype html>
<html lang="es">
    <head>
        <meta charset="utf-8">
        <link rel="stylesheet" href="css/xterm.css" type="text/css"/>
        <link rel="stylesheet" href="css/style.css" type="text/css"/>
        <script src="js/xterm.js"></script>
        <script src="js/socket.io.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/xterm/3.14.5/xterm.min.js"></script>
        <script src="AceEditor/src-min-noconflict/ace.js"></script>
        <script src="AceEditor/src-min-noconflict/ext-language_tools.js"></script>
        <title>C Online Compiler</title>
    </head>
    <body>
        <header id="header">
            <img class="logo" src="static/img/logo/encode_logo_light.png" alt="Logo Encode">
        </header>
        <section id="compiler">
            <nav id="nav_editor" class="nav_compiler">
                <button class="nav_button" onclick="sendToCompiler()">Compilar</button>
            </nav>
            <nav id="nav_terminal" class="nav_compiler">
                <button class="nav_button" onclick="resetTerminal()">Reiniciar</button>
                <button class="nav_button" onclick="clearTerminal()">Limpiar</button>
            </nav>
            <div id="editor"></div>
            <div id="terminal"></div>
            <div id="arguments_area">
                <label for="arguments">Argumentos:</label>
                <textarea id="arguments" name="arguments" rows="1" cols="60"></textarea>
            </div>
        </section>
        <footer id="footer">
            Trabajo de Fin de Grado (2021). Alejandro Asenjo Gómez
        </footer>
        <script src="js/ptyController.js"></script>
        <script src="js/aceEditor.js"></script>
    </body>
</html>
```

A.5. Hoja de estilo *Scss*

Código A.6: Hoja de estilo *Scss* de la web del compilador.

```
$base-color: #292929;
$border-dark: rgba($base-color, 0.88);

#header{
    background-color: #182233;
    box-shadow: 0px 0px 5px #999;
    padding-top: 10px;
    padding-bottom: 10px;
    padding-left: 13px;
    padding-right: 13px;
    height: 3.5em;

    .logo {
        width: auto;
        max-height: 100%;
    }
}

body{
    padding: 0;
    margin: 0;
    display: flex;
    flex-direction: column;
    min-height: 90vh;
    max-width: 100%;
    overflow-x: hidden;
}

section {
    background-color: aqua;
    flex-grow: 1; /* Expand the editor section the most */
}
```

Código A.7: Hoja de estilo *Scss* de la web del compilador.

```
#compiler {
    display: grid;
    width: 100 %;
    height: 100 %;
    grid-template-areas: "nav_editor nav_terminal"
                         "editor terminal"
                         "arguments_area arguments_area";
    grid-template-rows: 50px 1fr 50px;
    grid-template-columns: 1fr 1fr;

    .nav_compiler {
        display: flex;
        justify-content: right;
        align-items: center;
        border: 2px solid $border-dark;
        background-color: #8ca0ff;

        &#nav_editor {
            grid-area: nav_editor;
        }

        &#nav_terminal {
            grid-area: nav_terminal;
        }
    }

    .nav_button {
        width: auto;
        min-height: 50 %;
        max-height: 80 %;
        margin-right: 2.5 %;
        background-color: white;
        border-radius: 8px;

        &:hover {
            box-shadow: 0 12px 16px 0 rgba(0,0,0,0.24), 0 17px 50px 0 rgba(0,0,0,0.19);
        }
    }

    > #editor {
        grid-area: editor;
        border: 2px solid $border-dark;
    }
```

Código A.8: Hoja de estilo *Scss* de la web del compilador.

```
    > #arguments_area {
        grid-area: arguments_area;
        background-color: #8ca0ff;
        display: flex;
        justify-content: left;
        align-items: center;
        padding-left: 2 %;
        border: 2px solid $border-dark;

        textarea {
            margin: 1 %;
            resize: none;
            border: 1px solid $border-dark;
        }
    }

    > #terminal {
        grid-area: terminal;
        background-color: #151942;
        width: 100 % !important;
        border: 2px solid $border-dark;

        > .terminal {
            width: inherit !important;

            .xterm-screen {
                width: inherit !important;
            }
        }
    }
}

footer{
    display:flex;
    justify-content: center;
    align-items: center;
    background-color: #182233;
    color: white;
}
```

A.6. Documento *ptyController.js*

Código A.9: Implementa las acciones que puede llevar a cabo el usuario desde la interfaz web y realiza la conexión con el servidor a través del *socket*.

```
const term = new Terminal({
    cursorBlink: "bar",
    rows: 42,
    cols: 85,
    scrollback: 100,
    theme: {
        background: "#151942",
        foreground: "white",
        cursor: "white"
    }
});

const socket = io.connect("compiler.example.com");

var currentLine = "";

term.open(document.getElementById("terminal"));

term.on("key", function (key, ev) {
    if(ev.keyCode === 13){    // Enter
        if(currentLine) {
            term.write('\x1b[2K\r');
            term.write("\n\r");
            socket.emit("stdin", currentLine);
        }
    }else if(ev.keyCode === 8) {   // BackSpace
        currentLine = currentLine.slice(0, -1);
        term.write("\b \b");
    }else{
        currentLine += key;
        term.write(key);
    }
});

term.on("paste", function (data) {
    currentLine += data;
    term.write(data);
});
```

Código A.10: Implementa las acciones que puede llevar a cabo el usuario desde la interfaz web y realiza la conexión con el servidor a través del *socket*.

```
socket.on("connect", function (event) {
    socket.emit('Hello Server!');
});

socket.on("stdout", function (data) {
    term.write(data + "\n\r");
    currentLine = "";
});

function sendToCompiler() {
    socket.emit("compile", {
        code: editor.getValue(),
        args: document.getElementById("arguments").value
    });
}

function resetTerminal(){
    socket.emit("reset", 'Reset');
    currentLine = "";
    term.clear();
    term.write("\x1b[2K\r");
}

function clearTerminal(){
    currentLine = "";
    term.clear();
    term.write("\x1b[2K\r");
    term.write("~ $\n\r");
}
```

A.7. Documento *aceEditor.js*

Código A.11: Configura el editor de la interfaz web.

```
// Trigger extension
ace.require("ace/ext/language_tools");

const editor = ace.edit("editor", {
    mode: "ace/mode/c_cpp"
});

setLigthMode();

editor.setOptions({
    enableBasicAutocompletion: true,
    enableSnippets: true,
    enableLiveAutocompletion: true
});

editor.setValue("#include <stdio.h>\n\rint main (int argc, char *argv[]){\n\r\tprintf(\"Hello
    World\");\n\r\treturn 0;\n\r}");

function setLigthMode() {
    editor.setTheme("ace/theme/eclipse");
}

function setDarkMode() {
    editor.setTheme("ace/theme/tomorrow_night");
}
```

B

Docker

En este apéndice, se presenta el fichero *Dockerfile*, con el que se constituye la imagen de *Docker* correspondiente al compilador en línea. La base de la que parte este código ha sido extraída del repositorio oficial de *Node.js* [33].

Código B.1: Fichero de configuración *Dockerfile*.

```
FROM alpine:3.13

ENV NODE_VERSION 15.14.0

RUN addgroup --gid 5000 compilerGroup \
    && adduser -D compiler -h /home/compiler -u 5000 \
        -G compilerGroup -s /bin/sh -g compiler

RUN apk add --no-cache \
        libstdc++ \
    && apk add --no-cache --virtual .build-deps \
        curl \
    && ARCH= && alpineArch="$(apk --print-arch)" \
        && case "${alpineArch##*-}" in \
            x86_64) \
                ARCH='x64' \
                CHECKSUM="5aefd9f12592e6ed7e7a1fe2696576cf3e19d42c6103abcc3347cab2e54b7fb3" \
                ;; \
            *) ;; \
        esac \
    && if [ -n "${CHECKSUM}" ]; then \
    set -eu; \
    curl -fsSLO --compressed "https://unofficial-builds.nodejs.org/download/release/v$NODE_VERSION/
        node-v$NODE_VERSION-linux-$ARCH-musl.tar.xz"; \
    echo "$CHECKSUM node-v$NODE_VERSION-linux-$ARCH-musl.tar.xz" | sha256sum -c -\
        && tar -xJf "node-v$NODE_VERSION-linux-$ARCH-musl.tar.xz" -C /usr/local --
            strip-components=1 --no-same-owner \
        && ln -s /usr/local/bin/node /usr/local/bin/nodejs; \
    else \
```

Código B.2: Fichero de configuración *Dockerfile*.

```
    echo "Building from source" \
    # backup build
    && apk add --no-cache --virtual .build-deps-full \
        binutils-gold \
        g++ \
        gcc \
        gnupg \
        libgcc \
        linux-headers \
        make \
        python3 \
    # gpg keys listed at https://github.com/nodejs/node#release-keys
    && for key in \
        4ED778F539E3634C779C87C6D7062848A1AB005C \
        94AE36675C464D64BAFA68DD7434390BDBE9B9C5 \
        74F12602B6F1C4E913FAA37AD3A89613643B6201 \
        71DCFD284A79C3B38668286BC97EC7A07EDE3FC1 \
        8FCCA13FEF1D0C2E91008E09770F7A9A5AE15600 \
        C4F0DFFF4E8C1A8236409D08E73BC641CC11F4C8 \
        C82FA3AE1CBEDC6BE46B9360C43CEC45C17AB93C \
        DD8F2338BAE7501E3DD5AC78C273792F7D83545D \
        A48C2BEE680E841632CD4E44F07496B3EB3C1762 \
        108F52B48DB57BB0CC439B2997B01419BD92F80A \
        B9E2F5981AA6E0CD28160D9FF13993A75599653C \
    ; do \
        gpg --batch --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys "$key" || \
        gpg --batch --keyserver hkp://ipv4.pool.sks-keyservers.net --recv-keys "$key" || \
        gpg --batch --keyserver hkp://pgp.mit.edu:80 --recv-keys "$key" ; \
    done \
    && curl -fsSLO --compressed
        "https://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION.tar.xz" \
    && curl -fsSLO --compressed
        "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
    && gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \
    && grep " node-v$NODE_VERSION.tar.xz\$" SHASUMS256.txt | sha256sum -c -\
    && tar -xf "node-v$NODE_VERSION.tar.xz" \
    && cd "node-v$NODE_VERSION" \
    && ./configure \
    && make -j$(getconf _NPROCESSORS_ONLN) V= \
    && make install \
    && apk del .build-deps-full \
    && cd .. \
    && rm -Rf "node-v$NODE_VERSION" \
    && rm "node-v$NODE_VERSION.tar.xz" SHASUMS256.txt.asc SHASUMS256.txt; \
fi \
&& rm -f "node-v$NODE_VERSION-linux-$ARCH-musl.tar.xz" \
&& apk del .build-deps \
# smoke tests
&& node --version \
&& npm --version
```

Código B.3: Fichero de configuración *Dockerfile*.

```
ENV YARN_VERSION 1.22.5

RUN apk add --no-cache --virtual .build-deps-yarn curl gnupg tar \
    && for key in \
        6A010C5166006599AA17F08146C2130DFD2497F5 \
    ; do \
        gpg --batch --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys "$key" || \
        gpg --batch --keyserver hkp://ipv4.pool.sks-keyservers.net --recv-keys "$key" || \
        gpg --batch --keyserver hkp://pgp.mit.edu:80 --recv-keys "$key" ; \
    done \
    && curl -fsSLO --compressed
         "https://yarnpkg.com/downloads/$YARN_VERSION/yarn-v$YARN_VERSION.tar.gz" \
    && curl -fsSLO --compressed
         "https://yarnpkg.com/downloads/$YARN_VERSION/yarn-v$YARN_VERSION.tar.gz.asc" \
    && gpg --batch --verify yarn-v$YARN_VERSION.tar.gz.asc yarn-v$YARN_VERSION.tar.gz \
    && mkdir -p /opt \
    && tar -xzf yarn-v$YARN_VERSION.tar.gz -C /opt/ \
    && ln -s /opt/yarn-v$YARN_VERSION/bin/yarn /usr/local/bin/yarn \
    && ln -s /opt/yarn-v$YARN_VERSION/bin/yarnpkg /usr/local/bin/yarnpkg \
    && rm yarn-v$YARN_VERSION.tar.gz.asc yarn-v$YARN_VERSION.tar.gz \
    && apk del .build-deps-yarn \
    # smoke test
    && yarn --version

RUN apk update && \
    apk add --no-cache --virtual .gyp \
        make \
        python3 \
        g++ \
    && apk add \
        gcc \
        libc-dev

# Create app directory
WORKDIR /srv/node/app

COPY . .

RUN npm install \
    && rm /srv/node/app/package.json /srv/node/app/package-lock.json \
    && apk del .gyp

EXPOSE 3000

ENV HOME /home/compiler

USER compiler

RUN cd ~compiler

CMD ["node", "/srv/node/app/src/server.js"]
```

C

Recursos de Kubernetes

En este apéndice, se mostrarán algunos de los recursos de *Kubernetes* a los que se ha hecho referencia. Además, se presentarán algunas imágenes de la interfaz proporcionada por los tableros de *Grafana* instalados.

C.1. Monitorización del NGINX Ingress Controller

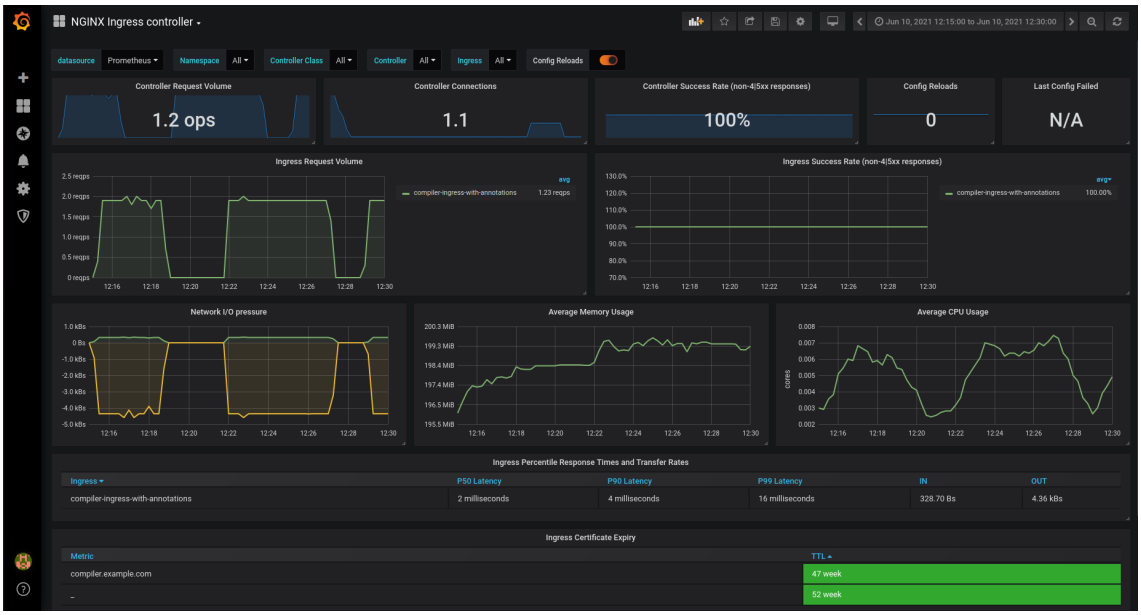

Figura C.1: Tablero de *Grafana* que monitoriza el *NGINX Ingress Controller*.

C.2. Monitorización del clúster

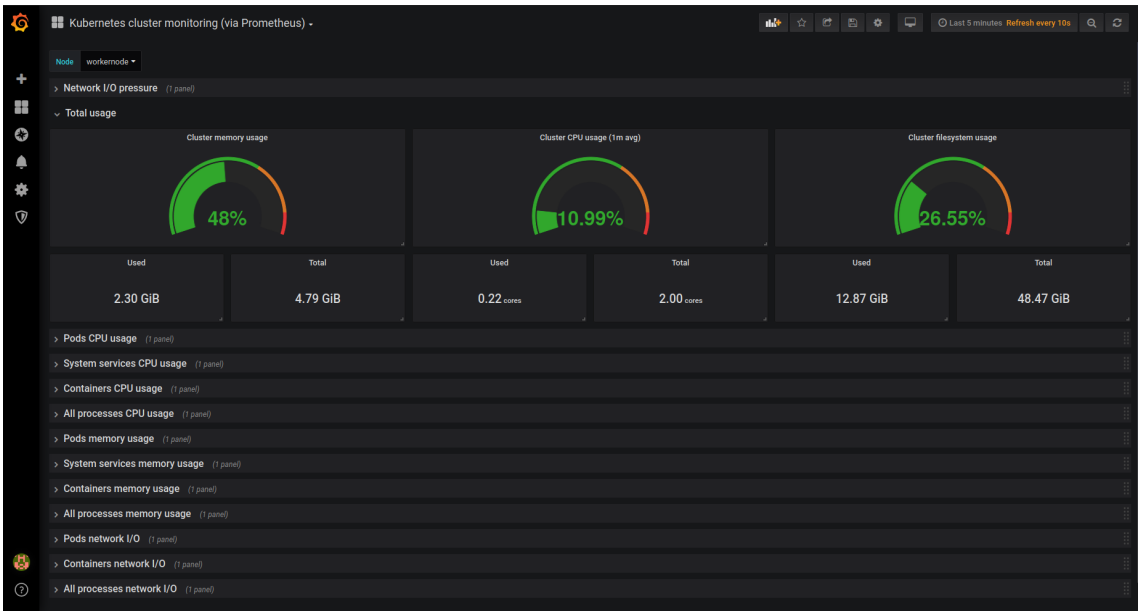

Figura C.2: Tablero de *Grafana* que muestra los recursos del *clúster*.

C.3. Network Policy

Código C.1: Fichero de configuración de la política de red de los *pods*.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: denegar-trafico-saliente
  namespace: compiler-app
spec:
  podSelector:
    matchLabels:
      networkpolicy: denegar-trafico-saliente
  egress: []
  policyTypes:
  -Egress
```


UAM
UNIVERSIDAD AUTONOMA
DE MADRID